Autori, în ordine alfabetică:

Adrian Niţă
profesor, Colegiul Naţional „Emanuil Gojdu", Oradea

Maria Niţă
profesor, Colegiul Naţional „Emanuil Gojdu", Oradea

Nicolae Olăroiu
profesor, Colegiul Naţional „B.P. Haşdeu", Buzău

Rodica Pintea
profesor, Liceul „Grigore Moisil", Bucureşti (capitolul 1)

Cristina Sichim
profesor, Colegiul Naţional „Ferdinand I", Bacău

Daniela Tarasă
Inspector Informatică, ISJ Bacău

Coordonatori:

Mihai Tătăran
cadru didactic asociat, Universitatea Politehnică Timişoara

Nuşa Dumitriu-Lupan
inspector General MECT

Petru Jucovschi
Developer Community Lead, Microsoft România

Microsoft

Suport de curs pentru elevi



Introducere în .Net Framework

Ediţia 2008

Microsoft .net Framework

Cerințe de sistem

Arhitectura suportată:
- x86
- x64 (WOW)

Sistem de operare suportat:
- Microsoft Windows XP
- Microsoft Windows Server 2003
- Windows Vista

Cerințe Hardware:
- Minimum: CPU 1.6 GHz, RAM 192 MB, Rezoluție Monitor 1024x768, Disc 5400 RPM
- Recomandat: CPU 2.2 GHz sau mai puternic, RAM 384 MB sau mai mult, rezoluție monitor 1280x1024, Disc 7200 RPM sau mai mult.
- Windows Vista: CPU 2.4 GHz, RAM 768 MB, Spațiu liber disc 1.3 GB pentru instalare completă

Resurse și Instrumente:
- www.microsoft.ro/ark - Academic Resource Kit, colecție de instrumente software și resurse educaționale.

Cuvânt Înainte

Dragi elevi,

Introducere în .NET Framework este un curs dezvoltat în cadrul programului Microsoft Parteneri pentru Educație, în colaborare cu un grup de profesori de informatică din România. Până la sfârșitul anului școlar 2007-2008 va fi disponibil în pe situl Microsoft România, în pagina Secțiuni pentru educație.

Cursul vă propune să explorați tehnologia **.NET**, cea mai răspândită platformă de aplicații software. Aveți posibilitatea să studiați soluții software și să dezvoltați aplicații ce pot fi trimise la concursuri sau pot fi integrate în proiecte educaționale.

Suportul de curs este publicat în două versiuni. Cea pentru elevi cuprinde doar componenta de specialitate. Versiunea pentru profesori cuprinde pe lângă componenta de specialitate și pe cea metodică de predare.

Suportul de curs poate fi descărcat gratuit și folosit exclusiv în procesul educațional.

Scopul acestei inițiative a programului ***Parteneri pentru Educație*** este de a încuraja dezvoltarea profesională a profesorilor și de a da un suflu nou experienței educaționale la materia Informatică.

Împreună cu partenerii, echipa Microsoft România vă mulțumește pentru interesul pentru studiul tehnologiei **.Net**.

Sperăm dragi elevi, să vă dezvoltați potențialul tehnic și creativ pentru a deveni competitivi după absolvirea liceului.

Sanda Foamete
SNR Manager de Proiecte Educaționale
Microsoft România

Autori, în ordine alfabetică:

Adrian Niță, profesor, Colegiul Național „Emanuil Gojdu”, Oradea
Maria Niță, profesor, Colegiul Național „Emanuil Gojdu”, Oradea
Nicolae Olăroiu, profesor Colegiul Național „B.P. Hașdeu”, Buzău
Rodica Pintea, profesor, Liceul „Grigore Moisil”, București (capitolul 1)
Cristina Sichim, profesor, Colegiul Național „Ferdinand I”, Bacău
Daniela Tarasă, Inspector Informatică, ISJ Bacău

Coordonatori:

Mihai Tătăran, cadru didactic asociat, Universitatea Politehnica din Timișoara
Nușa Dumitriu-Lupan, Inspector General MECT
Petru Jucovschi, Developer Community Lead, Microsoft România

Formatul electronic al textului digital: PDF

Editat de BYBLOS SRL sub coordonarea Agora Media SA, pentru Microsoft România. Ediția 2008.

ISBN: 973-86699-5-2

Notă:
Acest suport de curs este destinat elevilor de la clasele matematică-informatică și matematică-informatică intensiv, care au optat în programa școlară, pentru variantele: Programare orientată obiect, Progamare vizuală cu C# și Programare web cu Asp.Net. Suportul de curs poate fi utilizat gratuit exclusiv în procesul de predare-învățare. Este interzisă utilizarea suportului de curs ***„Introducere în .Net Framework”*** *pentru scopuri comerciale sau în alte scopuri în afara celui descris mai sus.*

CUPRINS

CAPITOLUL 1

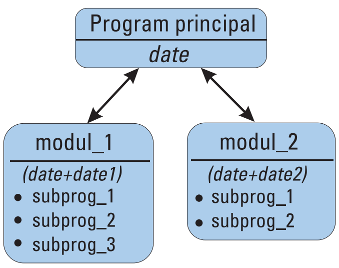
Programarea Orientată Obiect (POO) cu C#

1.1 Evolutia tehnicilor de programare

• **Programarea nestructurată** (un program simplu, ce utilizează numai variabile globale); complicațiile apar când prelucrarea devine mai amplă, iar datele se multiplică și se diversifică.

• **Programarea procedurală** (program principal deservit de subprograme cu parametri formali, variabile locale și apeluri cu parametri efectivi); se obțin avantaje privind depanarea și reutilizarea codului și se aplică noi tehnici privind transferul parametrilor și vizibilitatea variabilelor; complicațiile apar atunci când la program sunt asignați doi sau mai mulți programatori care nu pot lucra simultan pe un același fișier ce conține codul sursă.

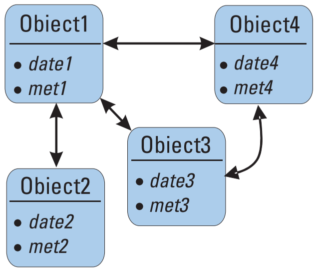
• **Programarea modulară** (gruparea subprogramelor cu funcționalități similare în module, implementate și depanate separat); se obțin avantaje privind independența și *încapsularea* (prin separarea zonei de implementare, păstrând vizibilitatea numai asupra zonei de interfață a modulului) și se aplică tehnici de asociere a procedurilor cu datele pe care le manevrează, stabilind și diferite reguli de acces la date și la subprograme.



Se observă că modulele sunt ”centrate” pe proceduri, acestea gestionând și setul de date pe care le prelucrează (*date+date1* din figură). Daca, de exemplu, dorim să avem mai multe seturi diferite de date, toate înzestrate comportamental cu procedurile din modulul `module1`, această arhitectură de aplicație nu este avantajoasă.

• **Programarea orientată obiect** (programe cu noi *tipuri* ce integrează atât datele, cât și metodele asociate creării, prelucrării și distrugerii acestor date); se obțin avantaje prin *abstractizarea* programării (programul nu mai este o succesiune de prelucrări, ci un ansamblu de obiecte care prind viață, au diverse proprietăți, sunt

capabile de acțiuni specifice și care interacționează în cadrul programului); intervin tehnici noi privind instanțierea, derivarea și polimorfismul tipurilor obiectuale.



1. 2 Tipuri de date obiectuale. Încapsulare

Un tip de date abstract (ADT) este o entitate caracterizată printr-o *structură de date* și un *ansamblu de operații* aplicabile acestor date. Considerând, în rezolvarea unei probleme de gestiune a accesului utilizatorilor la un anumit site, tipul abstract *USER*, vom obseva că sunt multe date ce caracterizează un utilizator Internet. Totuși se va ține cont doar de datele semnificative pentru problema dată. Astfel, "culoarea ochilor" este irelevantă în acest caz, în timp ce "data nașterii" poate fi importantă. În aceeași idee, operații specifice ca "se înregistrează", "comandă on-line" pot fi relevante, în timp ce operația "manâncă" nu este, în cazul nostru. Evident, nici nu se pun în discuție date sau operații nespecifice ("numărul de laturi" sau acțiunea "zboară").

Operațiile care sunt accesibile din afara entității formează *interfața* acesteia. Astfel, operații interne cum ar fi conversia datei de naștere la un număr standard calculat de la 01.01.1900 nu fac parte din interfața tipului de date abstract, în timp ce operația "plasează o comandă on-line" face parte, deoarece permite interacțiunea cu alte obiecte (SITE, STOC etc.)

O *instanță* a unui tip de date abstract este o "concretizare" a tipului respectiv, formată din valori efective ale datelor.

Un *tip de date obiectual* este un tip de date care implementează un tip de date abstract. Vom numi operațiile implementate în cadrul tipului de date abstract *metode*. Spunem că datele și metodele sunt *membrii* unui tip de date obiectual. Folosirea unui astfel de tip presupune: existența definiției acestuia, apelul metodelor și accesul la date.

Un exemplu de-acum clasic de tip de date abstract este STIVA. Ea poate avea ca date: numerele naturale din stivă, capacitatea stivei, vârful etc. Iar operațiile specifice pot fi: introducerea în stivă (*push*) și extragerea din stivă (*pop*). La implementarea tipului STIVA, vom defini o structura de date care să rețină valorile memorate în stivă și câmpuri de date simple pentru: capacitate, număr de elemente etc. Vom mai defini metode (subprograme) capabile să creeze o stivă vidă, care să introducă o valoare în stivă, să extragă valoarea din vârful stivei, să testeze dacă stiva este vidă sau dacă stiva este plină etc.

Crearea unei instanțe noi a unui tip obiectual, presupune operații specifice de "construire" a noului obiect, metoda corespunzătoare purtând numele de *constructor*. Analog, la desființarea unei instanțe și eliberarea spațiului de memorie

aferent datelor sale, se aplică o metodă specifică numită *destructor*[1].

O aplicație ce utilizează tipul obiectual STIVA, va putea construi două sau mai multe stive (de cărți de joc, de exemplu), le va umple cu valori distincte, va muta valori dintr-o stivă în alta după o anumită regulă desființând orice stivă golită, până ce rămâne o singură stivă. De observat că toate aceste prelucrări recurg la datele, constructorul, destructorul și la metodele din interfața tipului STIVA descris mai sus.

Principalul tip obiectual întâlnit în majoritatea mediilor de dezvoltare (Viisual Basic, Delphi, C++, Java, C#) poartă numele de clasă (*class*). Există și alte tipuri obiectuale (*struct*, *object*). O instanță a unui tip obiectual poartă numele de *obiect*.

La implementare, datele și metodele asociate trebuie să fie complet și corect definite, astfel încât utilizatorul să nu fie nevoit să țină cont de detalii ale acestei implementări. El va accesa datele, prin intermediul proprietăților și va efectua operațiile, prin intermediul metodelor puse la dispoziție de tipul obiectual definit. Spunem că tipurile de date obiectuale respectă principiul *încapsulării*. Astfel, programatorul ce utilizează un tip obiectual CONT (în bancă) nu trebuie să poarte grija modului cum sunt reprezentate în memorie datele referitoare la un cont sau a algoritmului prin care se realizează actualizarea soldului conform operațiilor de depunere, extragere și aplicare a dobânzilor. EL va utiliza unul sau mai multe conturi (instanțe ale tipului CONT), accesând proprietățile și metodele din interfață, realizatorul tipului obiectual asumându-și acele griji în momentul definirii tipului CONT.

Permițând extensia tipurilor de date abstracte, clasele pot avea la implementare:

- date și metode caracterisitice fiecărui obiect din clasă (membri de tip instanță),
- date și metode specifice clasei (membri de tip clasă).

Astfel, clasa STIVA poate beneficia, în plus, și de date ale clasei cum ar fi: numărul de stive generate, numărul maxim sau numărul minim de componente ale stivelor existente etc. Modificatorul `static` plasat la definirea unui membru al clasei face ca acela să fie un membru de clasă, nu unul de tip instanță. Dacă în cazul membrilor nestatici, există câte un exemplar al membrului respectiv pentru fiecare instanță a clasei, membrii statici sunt unici, fiind accesați în comun de toate instanțele clasei. Mai mult, membrii statici pot fi referiți chiar și fără a crea vreo instanță a clasei respective.

1.3. Supraîncărcare

Deși nu este o tehnică specifică programării orientată obiect, ea creează un anumit context pentru metodele ce formează o clasă și modul în care acestea pot fi (ca orice subprogram) apelate.

Prin supraîncarcare se înțelege posibilitatea de a defini în același domeniu de vizibilitate[2] mai multe funcții cu același nume, dar cu parametri diferiti ca tip și/sau ca număr. Astfel ansamblul format din numele funcției și lista sa de parametri reprezintă o modalitate unică de identificare numită *semnătură* sau amprentă. Supra-

[1] Datorită tehnicii de supraîncărcare C++, Java și C# permit existența mai multor constructori

[2] Noțiunile generale legate de vizibilitate se consideră cunoscute din programarea procedurală. Aspectele specifice și modificatorii de acces/vizibilitate pot fi studiați din documentațiile de referință C#.

încărcarea permite obținerea unor efecte diferite ale apelului în contexte diferite[3].

Apelul unei funcții care beneficiază, prin supraîncărcare, de două sau mai multe semnături se realizează prin selecția funcției a cărei semnătură se potrivește cel mai bine cu lista de parametri efectivi (de la apel).

Astfel, poate fi definită metoda ”comandă on-line” cu trei semnături diferite:

`comanda_online(cod_prod)` cu un parametru întreg (desemnând comanda unui singur produs identificat prin `cod_prod`.

`comanda_online(cod_prod,cantitate)` cu primul parametru întreg și celalalt real

`comanda_online(cod_prod,calitate)` cu primul parametru întreg și al-II-lea caracter.

1.4. Moștenire

Pentru tipurile de date obiectuale *class* este posibilă o operație de extindere sau specializare a comportamentului unei clase existente prin definirea unei clase noi ce moștenește datele și metodele clasei de bază, cu această ocazie putând fi redefiniți unii membri existenți sau adăugați unii membri noi. Operația mai poartă numele de *derivare*.

Clasa din care se moșteneștea se mai numește clasă *de bază* sau *superclasă*. Clasa care moștenește se numește *subclasă*, clasă derivată sau clasă descendentă.

Ca și în Java, în C# o subclasă poate moșteni de la o singură superclasă, adică avem de-a face cu moștenire simplă; aceeași superclasă însă poate fi derivată în mai multe subclase distincte. O subclasă, la randul ei, poate fi superclasă pentru o altă clasă derivată. O clasă de bază impreună cu toate clasele descendente (direct sau indirect) formeaza o ierarhie de clase. În C#, toate clasele moștenesc de la clasa de bază `Object`.

În contextul mecanismelor de moștenire trebuie amintiți modificatorii `abstract` și `sealed` aplicați unei clase, modificatori ce obligă la și respectiv se opun procesului de derivare. Astfel, o clasă abstractă trebuie obligatoriu derivată, deoarece direct din ea nu se pot obține obiecte prin operația de instanțiere, în timp ce o clasă sigilată (*sealed*) nu mai poate fi derivată (e un fel de terminal în ierarhia claselor). O metodă abstractă este o metodă pentru care nu este definită o implementare, aceasta urmând a fi realizată în clasele derivate din clasa curentă[4]. O metodă sigilată nu mai poate fi redefinită în clasele derivate din clasa curentă.

1.5. Polimorfism. Metode virtuale

Folosind o extensie a sensului etimologic, un obiect polimorfic este cel capabil să ia diferite forme, să se afle în diferite stări, să aibă comportamente diferite. Polimorfismul obiectual[5] se manifestă în lucrul cu obiecte din clase aparținând unei ierarhii de clase, unde, prin redefinirea unor date sau metode, se obțin membri dife-

[3] Capacitatea unor limbaje (este și cazul limbajului C#) de a folosi ca "nume" al unui subprogram un operator, reprezintă supraîncărcarea operatorilor. Aceasta este o facilitate care "reduce" diferențele dintre operarea la nivel abstract (cu DTA) și apelul metodei ce realizează acestă operație la nivel de implementare obiectuală. Deși ajută la sporirea expresivității codului, prin supraîncărcarea operatorilor și metodelor se pot crea și confuzii.

[4] care trebuie să fie și ea abstractă (virtuală pură, conform terminologiei din C++)

[5] deoarece tot aspecte polimorfice îmbracă și unele tehnici din programarea clasică sau tehnica supraîncărcărcării funcțiilor și operatorilor.

riți având însă același nume. Astfel, în cazul unei referiri obiectuale, se pune problema stabilirii datei sau metodei referite. Comportamentul polimorfic este un element de flexibilitate care permite stabilirea contextuală, în mod dinamic[6], a membrului referit.

De exemplu, dacă este definită clasa numită PIESA (de șah), cu metoda nestatică `muta(pozitie_initiala,pozitie_finala)`, atunci subclasele TURN și PION trebuie să aibă metoda `muta` definită în mod diferit (pentru a implementa maniera specifică a pionului de a captura o piesă ”en passant”[7]). Atunci, pentru un obiect T, aparținând claselor derivate din PIESA, referirea la metoda `muta` pare nedefinită. Totuși mecanismele POO permit stabilirea, în momentul apelului, a clasei proxime căreia îi aparține obiectul T și apelarea metodei corespunzătore (mutare de pion sau tură sau altă piesă).

Pentru a permite acest mecanism, metodele care necesită o decizie contextuală (în momentul apelului), se declară ca metode virtuale (cu modificatorul `virtual`). În mod curent, în C# modificatorului `virtual` al funcției din clasa de bază, îi corespunde un specificator `override` al funcției din clasa derivată ce redefinește funcția din clasa de bază.

O metodă ne-virtuală nu este polimorfică și, indiferent de clasa căreia îi aparține obiectul, va fi invocată metoda din clasa de bază.

1.6. Programare orientată obiect în C#

C# permite utilizarea OOP respectând toate principiile enunțate anterior.

Toate componentele limbajului sunt într-un fel sau altul, asociate noțiunii de clasă. Programul însuși este o clasă având metoda statică `Main()` ca punct de intrare, clasă ce nu se instanțiază. Chiar și tipurile predefinite `byte`, `int` sau `bool` sunt clase sigilate derivate din clasa `ValueType` din spațiul `System`. Pentru a evita unele tehnici de programare periculoase, limbajul oferă tipuri speciale cum ar fi: interfețe și delegări. Versiunii 2.0 a limbajului i s-a adăugat un nou tip: clasele generice[8],

1.7. Declararea unei clase

Sintaxa[9]: [atrib]$_o$ [modificatori]$_o$ **`class`** [nume_clasă][:clasa_de_bază]$_o$ [corp_clasă]$_o$

Atributele reprezintă informații declarative cu privire la entitatea definită.

Modificatorii reprezintă o secvență de cuvinte cheie dintre: `public protected internal private` (modificatori de acces) `new abstract sealed` (modificatori de moștenire)

Clasa de bază este clasa de la care moștenește clasa curentă și poate exista o singură astfel de clasă de bază. Corpul clasei este un bloc de declarări ale membrilor clasei: *constante* (valori asociate clasei), *câmpuri* (variabile), *tipuri* de date definite de

---

[6] Este posibil doar în cazul limbajelor ce permit "legarea întârziată". La limbajele cu "legare timpurie", adresa la care se face un apel al unui subprogram se stabilește la compilare. La limbajele cu legare întârziată, această adresa se stabileste doar in momentul rulării, putându-se calcula distinct, în funcție de contextul în care apare apelul.

[7] Într-o altă concepție, metoda muta poate fi implementată la nivelul clasei PIESA și redefinită la nivelul subclasei PION, pentru a particulariza acest tip de deplasare care capturează piesa peste care trece pionul în diagonală.

[8] echivalentrul claselor template din C++

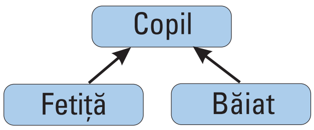
[9] [] din definiția schematică semnifică un neterminal, iar o semnifică o componentă opțională

utilizator, *metode* (subprograme), *constructori*, un *destructor*, *proprietăți* (caracteristici ce pot fi consultate sau setate), *evenimente* (instrumente de semnalizare), *indexatori* (ce permit indexarea instanțelor din cadrul clasei respective) și operatori.

- constructorii și destructorul au ca nume numele clasei proxime din care fac parte[10]
- metodele au nume care nu coincid cu numele clasei sau al altor membri (cu excepția metodelor, conform mecanismului de supraîncărcare)
- metodele sau constructorii care au același nume trebuie să difere prin semnătură[11]
- se pot defini date și metode statice (caracteristice clasei) și un constructor static care se execută la inițializarea clasei propriu-zise; ele formează un fel de ”context” al clasei
- se pot defini date și metode nestatice (de instanță) care se multiplică pentru fiecare instanță în parte în cadrul operației de instanțiere; ele formează contextele tuturor instanțelor clasei respective

Exemplul următor definește o ierarhie de clase (conform figurii alăturate)

```
public abstract class Copil { }
public class Fetita: Copil { }
public sealed class Baiat: Copil { }
```



Modificatorul `abstract` este folosit pentru a desemna faptul că nu se pot obține obiecte din clasa Copil, ci numai din derivatele acesteia (`Fetita`, `Baiat`), iar modificatorul `sealed` a fost folosit pentru a desemna faptul că nu se mai pot obtine clase derivate din clasa `Baiat` (de exemplu, subclasele `Baiat_cuminte` și `Baiat_rau`)

1.8. Constructori

Sintaxa:

[atrib]$_o$ [modificatori]$_o$ [nume_clasă] ([listă_param_formali]$_o$) [:inițializator]$_o$ [corp_constr]$_o$

Modificatori: `public protected internal private extern`

Inițializator: **`base`**([listă_param]$_o$), **`this`**([listă_param]$_o$) ce permite invocarea unui constructor anume[12] înainte de executarea instrucțiunilor ce formează corpul constructorului curent. Dacă nu este precizat niciun inițializator, se asociază implicit inițializatorul `base()`.

Corpul constructorului este format din instrucțiuni care se execută la crearea unui

[10] având în vedere că ele pot să facă parte dintr-o clasă interioară altei clase
[11] din semnătură nefăcând parte specificatorii ref și out asociați parametrilor
[12] Din clasa de bază (base) sau din clasa insăși (this)

nou obiect al clasei respective (sau la crearea clasei, în cazul constructorilor cu modificatorul `static`).

- pot exista mai mulți constructori care se pot diferenția prin lista lor de parametri
- constructorii nu pot fi moșteniți
- dacă o clasă nu are definit niciun constructor, se va asigna automat constructorul fără parametri al clasei de bază (clasa `object`, dacă nu este precizată clasa de bază)

Instanțierea presupune declararea unei variabile de tipul clasei respective și inițializarea acesteia prin apelul constructorului clasei (unul dintre ei, dacă sunt definiți mai mulți) precedat de operatorul `new`. Acestea se pot realiza și simultan într-o instrucțiune de felul:

[Nume_clasă] [nume_obiect]=**new** [Nume_clasă] ([listă_param]$_O$)

→ Utilizarea unui constructor fără parametri și a constructorului implicit în clasă derivată

```
public abstract class Copil
{ protected string nume;
   public Copil() {nume = Console.ReadLine();} //la inițializarea obiectului se 
citește
      //de la tastatură un șir de caractere ce va reprezenta numele copilului
}
class Fetita:Copil {}
...
Fetita f=new Fetita();
Copil c= new Copil(); //Pentru clasa Copil abstractă, s-ar fi obținut eroare aici
```

→ Supraîncărcarea constructorilor și definirea explicită a constructorilor în clase derivate

```
public class Copil
{ protected string nume; //dată accesibilă numai în interiorul clasei și 
claselor derivate
   public Copil() {nume = Console.ReadLine();}
   public Copil(string s) {nume=s;}
}
class Fetita:Copil
{ public Fetita(string s):base(s) {nume="Fetita "+nume}[13]
   public Fetita(){} //preia constructorul fără parametri din clasa de bază[14]
   //public Fetita(string s):base() {nume=s}
}
...
Copil c1= new Copil();    //se citeste numele de la tastatură
Copil c2= new Copil("Codrina");
Fetita f1=new Fetita();Fetita f2=new Fetita("Ioana");
```

[13] Preia și specializează constructorul al doilea din clasa de bază
[14] Este echivalent cu public Fetita():base(){}

Există două motive pentru care definiția constructorului al treilea din clasa Fetita este greșită și de aceea este comentată. Care sunt aceste motive?

1.9. Destructor

Sintaxa: [atrib]$_O$ [**`extern`**]$_O$ ~[nume_clasă] () [corp_destructor]$_O$

Corpul destructorului este format din instrucțiuni care se execută la distrugerea unui obiect al clasei respective. Pentru orice clasă poate fi definit un singur constructor. Destructorii nu pot fi moșteniți. În mod normal, destructorul nu este apelat în mod explicit, deoarece procesul de distrugere a unui obiect este invocat și gestionat automat de Garbagge Collector.

1.10. Metode

Sintaxa:[atrib]$_O$[modificatori]$_O$[tip_returnat] [nume] ([listă_param_formali]$_O$) [corp_metoda]$_O$

Modificatori: `new` `public` `protected` `internal` `private` `static` `virtual` `abstract` `sealed` `override` `extern`[15]

Tipul rezultat poate fi un tip definit sau `void`. Numele poate fi un simplu identificator sau, în cazul în care definește în mod explicit un membru al unei interfețe, numele este de forma [nume_interfata].[nume_metoda]

Lista de parametri formali este o succesiune de declarări despărțite prin virgule, declararea unui parametru având sintaxa: [atrib]$_O$ [modificator]$_O$ [tip] [nume]

Modificatorul unui parametru poate fi `ref` (parametru de intrare și ieșire) sau `out` (parametru care este numai de ieșire). Parametrii care nu au niciun modificator sunt parametri de intrare.

Un parametru formal special este parametrul tablou cu sintaxa:

[atrib]$_O$ **`params`** [tip]**`[]`** [nume].

Pentru metodele abstracte și externe, corpul metodei se poate reduce la un semn **;**

Semnătura fiecărei metode este formată din numele metodei, modificatorii acesteia, numărul și tipul parametrilor[16]

Numele metodei trebuie să difere de numele oricărui alt membru care nu este metodă.

La apelul metodei, orice parametru trebuie să aibă același modificator ca la definire

Invocarea unei metode se realizează prin sintagma [nume_obiect].[nume_metoda] (pentru metodele nestatice) și respectiv [nume_clasă].[nume_metoda] (pentru metodele statice).

Definirea datelor și metodelor **statice** corespunzătoare unei clase

---

15 Poate fi folosit cel mult unul dintre modificatorii static, virtual șI override ; nu pot apărea împreună new și override, abstract nu poate să apară cu niciunul dintre static, virtual, sealed, extern; private nu poate să apară cu niciunul dintre virtual, override și abstract; seald obligă și la override

16 Din semnătură (amprentă) nu fac parte tipul returnat, numele parametrilor formali și nici specificatorii ref și out.

```
    public class Copil
    { public const int nr_max = 5;     //constantă
      public static int nr_copii=0;    //câmp simplu (variabilă)
      static Copil[] copii=new Copil[nr_max]; //câmp de tip tablou (variabilă)
      public static void adaug_copil(Copil c) //metodă
      { copii[nr_copii++] = c;
        if (nr_copii==nr_max) throw new Exception("Prea multi copii");
      }
      public static void afisare()     //metodă
      {
          Console.WriteLine("Sunt {0} copii:", nr_copii);
          for (int i = 0; i<nr_copii; i++)
              Console.WriteLine("Nr.{0}. {1}", i+1, copii[i].nume);
      } ...17
    }
...
Fetita c = new Fetita();Copil.adaug_copil(c);
```

referința noului obiect se memorează în tabloul static `copii` (caracteristic clasei) și se incrementează data statică `nr_copii`

```
Baiat c = new Baiat(); Copil.adaug_copil(c);
Copil c = new Copil(); Copil.adaug_copil(c);
Copil.afisare(); //se afișează o listă cu numele celor 3 copii
```

Definirea datelor și metodelor **nestatice** corespunzătoare clasei Copil și claselor derivate

```
public class Copil
    { ...
      public string nume;
      public virtual void se_joaca()   //virtual à se poate suprascrie la
derivare
        {Console.WriteLine("{0} se joaca.", this.nume);}
      public void se_joaca(string jucaria)    //nu permite redefinire18
        {Console.WriteLine("{0} se joaca cu {1}.", this.nume, jucaria);}
    }     //supraîncărcarea metodei se_joaca
class Fetita:Copil
    {   public override void se_joaca() //redefinire à comportament polimorfic
          {Console.WriteLine("{0} leagana papusa.",this.nume);}
    }
class Baiat:Copil
    {   public override void se_joaca()
            {Console.WriteLine("{0} chinuie pisica.",this.nume);}
    }
...
```

[17] Se are în vedere și constructorul fără parametri definit și preluat implicit în subclasele din cadrul primului exemplu din subcapitolul 1.8: `public Copil() {nume = Console.ReadLine();}`

[18] Decât cu ajutorul modificatorului new pentru metoda respectivă în clasa derivată

```
Fetita c = new Fetita();c.se_joaca("pisica");c.se_joaca();    //polimorfism
Baiat c = new Baiat();c.se_joaca("calculatorul");c.se_joaca();      //polimorfism
Copil c = new Copil();c.se_joaca();     //polimorfism
```

Pentru a evidenția mai bine comportamentul polimorfic, propunem secvența următoare în care nu se știe exact ce este obiectul copii[i] (de tip Copil, Fetita sau Baiat):

```
for (int i=0; i<nr_copii; i++) copii[i].se_joaca();
```

1.11. Proprietăți

Proprietatea este un membru ce permite accesul controlat la datele-membru ale clasei.

Sintaxa: [atrib]$_O$ [modificatori]$_O$ [tip] [nume_proprietate] {[metode_de_acces]$_O$}

Observațiile privind modificatorii și numele metodelor sunt valabile și în cazul proprietăților.

Metodele de acces sunt două: `set` și `get`. Dacă proprietatea nu este abstractă sau externă, poate să apară una singură dintre cele două metode de acces sau amândouă, în orice ordine.

Este o manieră de lucru recomandabilă aceea de a proteja datele membru (câmpuri) ale clasei, definind instrumente de acces la acestea: pentru a obține valoarea câmpului respectiv (`get`) sau de a memora o anumită valoare în câmpul respectiv (`set`). Dacă metoda de acces `get` este perfect asimilabilă cu o metodă ce retunează o valoare (valoarea datei pe care vrem s-o obținem sau valoarea ei modificată conform unei prelucrări suplimentare specifice problemei în cauză), metoda `set` este asimilabilă cu o metodă care un parametru de tip valoare (de intrare) și care atribuie (sau nu, în funcție de context) valoarea respectivă câmpului. Cum parametrul corespunzător valorii transmise nu apare în structura sintactică a metodei, este de stiut că el este implicit identificat prin cuvântul `value`. Dacă se supune unor condiții specifice problemei, se face o atribuire de felul câmp=`value`.

Definirea în clasa Copil a proprietății Nume, corespunzătoare câmpului protejat ce reține, sub forma unui șir de caractere, numele copilului respctiv. Se va observă că proprietatea este moștenită și de clasele derivate Fetita și Băiat[19].

```
public class Copil
{...
    string nume; // este implicit protected
    public string Nume //proprietatea Nume
    {  get
       { if(char.IsUpper(nume[0]))return nume; else return nume.ToUpper();}
       set { nume = value; }
    }
```

[19] Desigur că proprietatea care controlează accesul la câmpul identificat prin nume se poate numi cu totul altfel (proprietatea Nume fiind ușor de confundat cu câmpul de date nume).

```
    public Copil() {Nume = Console.ReadLine();}       //metoda set
}
class Fetita:Copil
{  public override void se_joaca()
     {Console.WriteLine("{0} leagana papusa.",this.Nume);}  //metoda get
}
```
[20]

1.12. Evenimente și delegări

Evenimentele sunt membri ai unei clase ce permit clasei sau obiectelor clasei să facă notificări, adică să anunțe celelalte obiecte asupra unor schimbări petrecute la nivelul stării lor. Clasa furnizoare a unui eveniment *publică* (pune la dispoziția altor clase) acest lucru printr-o declarare `event` care asociază evenimentului un *delegat*, adică o referință către o funcție necunoscută căreia i se precizează doar antetul, funcția urmând a fi implementată la nivelul claselor interesate de evenimentul respectiv. Este modul prin care se realizează comunicarea între obiecte. Tehnica prin care clasele implementează metode (*handler-e*) ce răspund la evenimente generate de alte clase poartă numele de *tratare a evenimentelor*.

Sintaxa: [atrib]$_O$ [modificatori]$_O$ **event** [tip_delegat] [nume]

Modificatorii permiși sunt aceiași ca și la metode.

Tipul delegat este un tip de date ca oricare altul, derivat din clasa sigilată `Delegate`, din spațiul `System`. Definirea unui tip delegat se realizează prin declararea:

[atrib]$_O$ [modificatori]$_O$ **delegate** [tip_rezultat] [nume_delegat] ([listă_param_formali]$_O$)

Un delegat se poate defini și în afara clasei generatoare de evenimente și poate servi și altor scopuri în afara tratării evenimentelor. Prezentăm în continuare un exemplu.

De exemplu, dacă dorim să definim o metodă asociată unui vector de numere întregi, metodă ce verifică dacă vectorul este o succesiune "bine aranjată" (orice două valori succesive respectă o anumită regulă), o implementare "generică" se poate realiza folosind delegări:

```
public delegate bool pereche_ok(object t1, object t2);
public class Vector
{   public const int nmax = 4;
    public int[] v=new int[nmax];
    public Vector()
    {  Random rand = new Random();
       for (int i = 0; i < nmax; i++) v[i] = rand.Next(0,5);
    }
    public void scrie()
```

[20] De observat că în exemplul anterior (subcapitolul 1.10), câmpul nume era declarat public, pentru a permite accesul "general" la câmpul respectiv de date. Iar metodele și constructorii foloseau identificatorul nume și nu proprietatea Nume.

```
    {  for (int i = 0; i < nmax; i++) Console.Write("{0}, ", v[i]);
       Console.WriteLine();
    }
    public bool aranj(pereche_ok ok)//ok e o delegare către o funcţie necunoscută
    {  for (int i = 0; i < nmax-1; i++)
             if (!ok(v[i], v[i + 1])) return false;
       return true;
    }
}
```

Dacă în clasa-program[21] se adugă funcţiile (exprimând două "reguli de aranjare" posibile)

```
public static bool f1(object t1, object t2)
  {if ((int)t1 >= (int)t2) return true;else return false;}
public static bool f2(object t1, object t2)
  {if ((int)t1 <= (int)t2) return true;else return false;}
```

atunci o secvenţă de prelucrare aplicativă ar putea fi:

```
static void Main(string[] args)
{  Vector x;
   do {
       x =new Vector();x.scrie();
       if (x.aranj(f1))Console.WriteLine("Monoton descrescator");
       if (x.aranj(f2))Console.WriteLine("Monoton crescator");
     } while (Console.ReadKey(true).KeyCar!='\x001B'); //Escape
}
```

Revenind la evenimente, descriem pe scurt un exemplu teoretic de declarare şi tratare a unui eveniment. În clasa `Vector` se consideră că interschimbarea valorilor a două componente ale unui vector e un eveniment de interes pentru alte obiecte sau clase ale aplicaţiei. Se defineşte un tip delegat `TD` (să zicem) cu nişte parametri de interes[22] şi un eveniment care are ca asociat un delegat `E` (de tip `TD`)[23]. Orice obiect `x` din clasa `Vector` are un membru `E` (iniţial *null*). O clasă `C` interesată să fie înştiinţată când se face vreo interschimbare într-un vector pentru a genera o animaţie, de exemplu, va implementa o metodă `M` ce realizează animaţia şi va adăuga pe `M` (prin intermediul unui delegat) la `x.E`[24]. Cumulând mai multe astfel de referinţe, `x.E` ajunge un fel de listă de metode (*handlere*). În clasa `Vector`, în metoda `sort`, la interschimbarea valorilor a două componente se invocă delegatul `E`. Invocarea lui `E` realizează de fapt activarea tuturor metodelor adăugate la `E`.

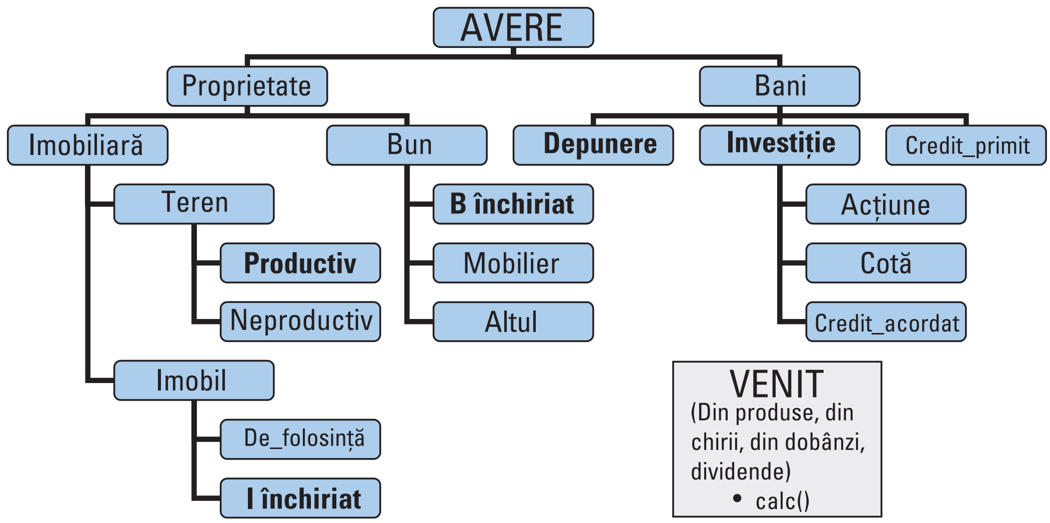
Care credeţi că sunt motivele pentru care apelăm la evenimente în acest caz, când pare mult mai simplu să apelăm direct metoda `M` la orice interschimbare?

21 Independent de definiţia clasei Vector
22 De exmplu indicii componentelor interschimbate
23 A se observa că evenimentul în sine este anonim, doar delegatul asociat are nume
24 într-o atribuire de felul x.E+=new [tip_delegat](M)

1.13. Interfețe

Interfețele sunt foarte importante în programarea orientată pe obiecte, deoarece permit utilizarea polimorfismului într-un sens mai extins.O interfață este o componentă a aplicației, asemănătoare unei clase, ce declară prin membrii săi (metode, proprietăți, evenimente și indexatori) un ”comportament” unitar aplicabil mai multor clase, comportament care nu se poate defini prin ierarhia de clase a aplicației.

De exemplu, dacă vom considera arborele din figura următoare, în care AVERE este o clasă abstractă, iar derivarea claselor a fost concepută urmărind proprietățile comune ale componentelor unei averi, atunci o clasă VENIT nu este posibilă, deoarece ea ar moșteni de la toate clasele evidențiate, iar moștenirea multiplă nu este admisă în C#.



Pentru metodele din cadrul unei interfețe nu se dă nici o implementare, ci sunt pur și simplu specificate, implementarea lor fiind furnizată de unele dintre clasele aplicației[25]. Nu există instanțiere în cazul interfețelor, dar se admit derivări, inclusiv moșteniri multiple.

În exemplul nostru, se poate defini o interfață VENIT care să conțină antetul unei metode `calc` (să zicem) pentru calculul venitului obținut, fiecare dintre clasele care implementează interfața VENIT fiind obligată să furnizeze o implementare (după o formulă de calcul specifică) pentru metoda `calc` din interfață. Orice clasă care dorește să adere la interfață trebuie să implementeze toate metodele din interfață. Toate clasele care moștenesc dintr-o clasă care implementează o interfață moștenesc, evident, metodele respective, dar le pot și redefini (de exemplu, clasa Credit_acordat redefinește metoda calc din clasa Investiție, deoarece formula de calcul implementată acolo nu i se ”potrivește” și ei[26]).

De exemplu, dacă presupunem că toate clasele subliniate implementează interfața VENIT, atunci pentru o avere cu acțiuni la două firme, un imobil închiriat și o

[25] Acele clase care "aderă" la o interfață spunem că "implementează" interfața respectivă
[26] Dacă în sens polimorfic spunem că Investiție este și de tip Bani și de tip Avere, tot așa putem spune că o clasă care implementează interfața VENIT și clasele derivate din ea sunt și de tip VENIT

depunere la bancă, putem determina venitul total:

```
Actiune act1 = new Actiune();Actiune act2 = new Actiune();
I_inchiriat casa = new I_inchiriat();Depunere dep=new Depunere();
Venit[] venituri = new Venit()[4];
venituri[0] = act1; venituri[1] = act2;
venituri[2] = casa; venituri[3] = dep;
...
int t=0;
for(i=0;i<4;i++) t+=v[i].calc();
```

Găsiți două motive pentru care interfața VENIT și rezovarea de mai sus oferă o soluție mai bună decât: `t=act1.calc()+act2.calc()+casa.calc()+dep.calc()`.

CAPITOLUL 2

Platforma .NET

2.1 Prezentare

.NET este un cadru (*Framework*) de dezvoltare software unitară care permite realizarea, distribuirea și rularea aplicațiilor-desktop Windows și aplicațiilor WEB.

Tehnologia .NET pune laolaltă mai multe tehnologii (ASP, XML, OOP, SOAP, WDSL, UDDI) și limbaje de programare (VB, C++, C#, J#) asigurând totodată atât portabilitatea codului compilat între diferite calculatoare cu sistem Windows, cât și reutilizarea codului în programe, indiferent de limbajul de programare utilizat.

.NET Framework este o componentă livrată împreună cu sistemul de operare Windows. De fapt, .NET 2.0 vine cu Windows Server 2003, se poate instala pe versiunile anterioare, până la Windows 98 inclusiv; .NET 3.0 vine instalat pe Windows Vista și poate fi instalat pe versiunile Windows XP cu SP2 și Windows Server 2003 cu minimum SP1.

Pentru a dezvolta aplicații pe platforma .NET este bine sa avem 3 componente esențiale:

- un set de limbaje (C#, Visual Basic .NET, J#, Managed C++, Smalltalk, Perl, Fortran, Cobol, Lisp, Pascal etc),
- un set de medii de dezvoltare (Visual Studio .NET, Visio),
- și o bibliotecă de clase pentru crearea serviciilor Web, aplicațiilor Web și aplicațiilor desktop Windows.

Când dezvoltăm aplicații .NET, putem utiliza:

- Servere specializate - un set de servere Enterprise .NET (din familia SQL Server 2000, Exchange 2000 etc), care pun la dispoziție funcții de stocare a bazelor de date, email, aplicații B2B (*Bussiness to Bussiness* — comerț electronic între partenerii unei afaceri).
- Servicii Web (în special comerciale), utile în aplicații care necesită identificarea utilizatorilor (de exemplu, .NET Passport - un mod de autentificare folosind un singur nume și o parolă pentru toate ste-urile vizitate)
- Servicii incluse pentru dispozitive non-PC (Pocket PC Phone Edition, Smartphone, Tablet PC, Smart Display, XBox, set-top boxes, etc.)

.NET Framework

Componenta .NET Framework stă la baza tehnologiei .NET, este ultima interfață

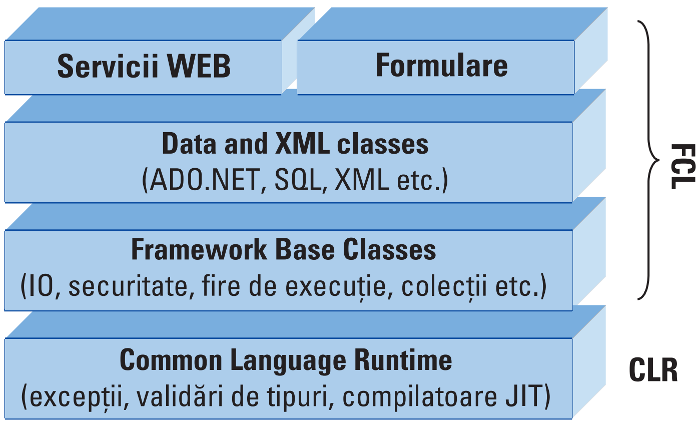
între aplicațiile .NET și sistemul de operare și actualmente conține:

Limbajele C#, VB.NET, C++ și J#. Pentru a fi integrate în platforma .NET toate aceste limbaje respectă niște specificații OOP numite *Common Type System* (CTS). Ele au ca elemente de bază: clase, interfețe, delegări, tipuri valoare și referință, iar ca mecanisme: moștenire, polimorfism și tratarea excepțiilor.

Platforma comună de executare a programelor numită *Common Language Runtime* (CLR), utilizată de toate cele 4 limbaje. CTS face parte din CLR.

Ansamblul de biblioteci necesare în realizarea aplicațiilor desktop sau Web numit *Framework Class Library* (FCL).

Arhitectura .NET Framework



Componenta .NET Framework este formată din compilatoare, biblioteci și alte executabile utile în rularea aplicațiilor .NET. Fișierele corespunzătoare se află, în general, în directorul C:\WINDOWS\Microsoft. NET\Framework\V2.0…. (corespunzător versiunii instalate)

2.2. Compilarea programelor

Un program scris într-unul dintre limbajele .NET conform *Common Language Specification* (CLS) este compilat în *Microsoft Intermediate Language* (MSIL sau IL). Codul astfel obținut are extensia *exe*, dar nu este direct executabil, ci respectă formatul unic MSIL.

CLR include o mașină virtuală asemănătoare cu o mașină Java, ce execută instrucțiunile IL rezultate în urma compilării. Mașina folosește un compilator special JIT (*Just In Time*). Compilatorul JIT analizează codul IL corespunzător apelului unei metode și produce codul mașină adecvat și eficient. El recunoaște secvențele de cod pentru care s-a obținut deja codul mașină adecvat permițând reutilizarea acestuia fără recompilare, ceea ce face ca, pe parcursul rulării, aplicațiile .NET să fie din ce în ce mai rapide.

Faptul că programul IL produs de diferitele limbaje este foarte asemănător are ca rezultat interoperabilitatea între aceste limbaje. Astfel, clasele și obiectele create într-un limbaj specific .NET pot fi utilizate cu succes în altul.

În plus, CLR se ocupă de gestionarea automată a memoriei (un mecanism implementat în platforma .NET fiind acela de eliberare automată a zonelor de memorie asociate unor date devenite inutile — *Garbage Collection*).

Ca un element de portabilitate, trebuie spus că .NET Framework este implementarea unui standard numit Common Language Infrastructure (http://www.ecma-international.org/publications/standards/Ecma-335.htm), ceea ce permite rularea aplicațiilor .NET, în afară de Windows, și pe unele tipuri de Unix, Linux, Solaris, Mac OS X și alte sisteme de operare (http://www.mono-project.com/Main_Page).

2.3. De ce am alege .NET?

În primul rând pentru că ne oferă instrumente pe care le putem folosi și în alte programe, oferă acces ușor la baze de date, permite realizarea desenelor sau a altor elemente grafice. Spațiul de nume System.Windows.Forms conține instrumente (controale) ce permit implementarea elementelor interfeței grafice cu utilizatorul. Folosind aceste controale, puteți proiecta și dezvolta rapid și interactiv, elementele interfeței grafice. Tot .NET vă oferă clase care efectuează majoritatea sarcinilor uzuale cu care se confruntă programele și care plictisesc și fură timpul programatorilor, reducând astfel timpul necesar dezvoltării aplicațiilor.

CAPITOLUL **3**

Limbajul C#

3.1. Caracterizare

Limbajul C# fost dezvoltat de o echipă restrânsă de ingineri de la Microsoft, echipă din care s-a evidențiat Anders Hejlsberg (autorul limbajului Turbo Pascal și membru al echipei care a proiectat Borland Delphi).

C# este un limbaj simplu, cu circa 80 de cuvinte cheie, și 12 tipuri de date predefinite. El permite programarea structurată, modulară și orientată obiectual, conform perceptelor moderne ale programării profesioniste.

Principiile de bază ale programării pe obiecte (ÎNCAPSULARE, MOȘTENIRE, POLIMORFISM) sunt elemente fundamentale ale programării C#. În mare, limbajul moștenește sintaxa și principiile de programare din C++. Sunt o serie de tipuri noi de date sau funcțiuni diferite ale datelor din C++, iar în spiritul realizării unor secvențe de cod sigure (*safe*), unele funcțiuni au fost adăugate (de exemplu, interfețe și delegări), diversificate (tipul *struct*), modificate (tipul *string*) sau chiar eliminate (moștenirea multiplă și pointerii către funcții). Unele funcțiuni (cum ar fi accesul direct la memorie folosind pointeri) au fost păstrate, dar secvențele de cod corespunzătoare se consideră ”nesigure”.

3.2. Compilarea la linia de comandă

Se pot dezvolta aplicații .NET și fără a dispune de mediul de dezvoltare Visual Studio, ci numai de .NET SDK (pentru 2.0 și pentru 3.0). În acest caz, codul se scrie în orice editor de text, fișierele se salvează cu extensia cs, apoi se compilează la linie de comandă.

Astfel, se scrie în Notepad programul:

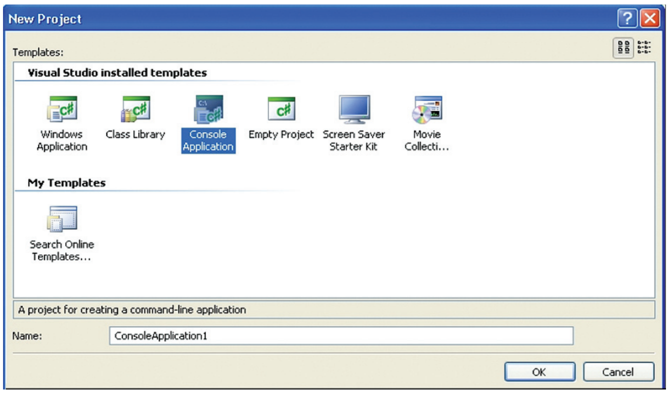
```
using System;
class primul
{
  static void Main()
  {   Console.WriteLine("Primul program");
      Console.ReadKey(true);
  }
}
```

Dacă se salvează fișierul **`primul.cs`**, în directorul **`WINDOWS\Microsoft.NET\Framework\V2.0`**, atunci scriind la linia de comandă: **`csc primul.cs`** se va obține fișierul **`primul.exe`** direct executabil pe o platformă .NET.

3.3. Crearea aplicațiilor consolă

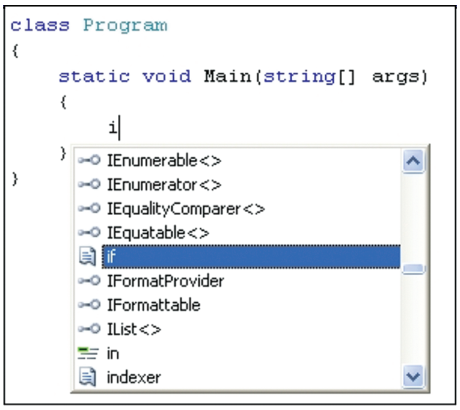
Pentru a realiza aplicații în mediul de dezvoltare Visual Studio, trebuie să instalăm o versiune a acestuia, eventual versiunea free **Microsoft Visual C# 2005/2008 Express Edition** de la adresa http://msdn.microsoft.com/vstudio/express/downloads/default.aspx. Pentru început, putem realiza aplicații consolă (ca și cele din Borland Pascal sau Borland C).

După lansare, alegem opțiunea *New Project* din meniul *File*. În fereastra de dialog (vezi figura), selectăm pictograma **Console Application**, după care, la **Name**, introducem numele aplicației noastre.

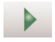


Fereastra în care scriem programul se numește implicit **Programs.cs** și se poate modifica prin salvare explicită (*Save As*). Extensia **cs** provine de la **C S**harp.

În scrierea programului suntem asistați de **IntelliSense**, ajutorul contextual.



Compilarea programului se realizează cu ajutorul opțiunii *Build Solution* (**F6**) din meniul Build. Posibilele erori de compilare sunt listate în fereastra *Error List*. Efectuând dublu clic pe fiecare eroare în parte, cursorul din program se poziționează pe linia conținând eroarea.

Rularea programului se poate realiza în mai multe moduri: rapid fără asistență de depanare (*Start Without Debugging* **Shift+F5**) , rapid cu asistență de depanare (*Start Debugging* **F5** sau cu butonul  din bara de instrumente), rulare pas cu pas (*Step Into* **F11** și *Step Over* **F12**) sau rulare rapidă până la linia marcată ca punct de întrerupere (*Toggle Breakpoint* **F9** pe linia respectivă și apoi *Start Debugging* **F5**). Încetarea urmăririi pas cu pas (*Stop Debugging* **Shift+F5**) permite ieșirea din modul depanare și revenirea la modul normal de lucru. Toate opțiunile de rulare și depanare se găsesc în meniul *Debug* al mediului.



Fereastra de cod și ferestrele auxiliare ce ne ajută în etapa de editare pot fi vizualizate alegând opțiunea corespunzătoare din meniul *View*. Ferestrele auxiliare utile în etapa de depanare se pot vizualiza alegând opțiunea corespunzătoare din meniul *Debug/Windows*.

3.4. Structura unui program C#

Să începem cu exemplul clasic “Hello World” adaptat la limbajul C#:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

O aplicatie C# este formată din una sau mai multe **clase**, grupate în **spații de nume** (**namespaces**). Un spațiu de nume cuprinde mai multe clase cu nume diferite având funcționalități înrudite. Două clase pot avea același nume cu condiția ca ele să fie definite în spații de nume diferite. În cadrul aceluiași spațiu de nume poate apărea definiția unui alt spațiu de nume, caz în care avem de-a face cu spații de nume imbri-

cate. O clasă poate fi identificată prin numele complet (nume precedat de numele spațiului sau spațiilor de nume din care face parte clasa respectivă, cu separatorul punct). În exemplul nostru, HelloWorld.Program este numele cu specificație completă al clasei Program.

O clasă este formată din date și metode (funcții). Apelarea unei metode în cadrul clasei în care a fost definită aceasta presupune specificarea numelui metodei. Apelul unei metode definite în interiorul unei clase poate fi invocată și din interiorul altei clase, caz în care este necesară specificarea clasei și apoi a metodei separate prin punct. Dacă în plus, clasa aparține unui spațiu de nume neinclus în fișierul curent, atunci este necesară precizarea tuturor componentelor numelui: *spațiu.clasă.metodă* sau *spațiu.spațiu.clasă.metodă* etc.

În fișierul nostru se află două spații de nume: unul definit (*HelloWorld*) și unul extern inclus prin directiva `using` (*System*). `Console.WriteLine` reprezintă apelul metodei *WriteLine* definită în clasa *Console*. Cum în spațiul de nume curent este definită doar clasa Program, deducem că definiția clasei *Console* trebuie să se găsească în spațiul *System*.

Pentru a facilita cooperarea mai multor programatori la realizarea unei aplicații complexe, există posibilitatea de a segmenta aplicația în mai multe fișiere numite **assemblies**. Într-un *assembly* se pot implementa mai multe spații de nume, iar părți ale unui aceeași spațiu de nume se pot regăsi în mai multe assembly-uri. Pentru o aplicație consolă, ca și pentru o aplicație Windows de altfel, este obligatoriu ca una (și numai una) dintre clasele aplicației să conțină un „punct de intrare” (*entry point*), și anume metoda (funcția) **Main**.

Să comentăm programul de mai sus:

linia 1: este o directivă care specifică faptul că se vor folosi clase incluse în spațiul de nume **System**. În cazul nostru se va folosi clasa **Console**.

linia 3: spațiul nostru de nume

linia 5: orice program C# este alcătuit din una sau mai multe clase

linia 7: metoda **Main**, „punctul de intrare” în program

linia 9: clasa **Console**, amintită mai sus, este folosită pentru operațiile de intrare/ieșire. Aici se apelează metoda **WriteLine** din acestă clasă, pentru afișarea mesajului dorit pe ecran.

3.5. Sintaxa limbajului

Ca și limbajul C++ cu care se înrudește, limbajul C# are un alfabet format din litere mari și mici ale alfabetului englez, cifre și alte semne. Vocabularul limbajului este format din acele ”simboluri”[27] cu semnificații lexicale în scrierea programelor: cuvinte (nume), expresii, separatori, delimitatori și comentarii.

Comentarii

comentariu pe un rând prin folosirea **//** Tot ce urmează după caracterele **//** sunt considerate, din acel loc, până la sfârșitul rândului drept comentariu

[27] Este un termen folosit un pic echivoc și provenit din traduceriea cuvântului "token"

```
// Acesta este un comentariu pe un singur rand
```

comentariu pe mai multe rânduri prin folosirea **/*** și ***/** Orice text cuprins între simbolurile menționate mai sus se consideră a fi comentariu. Simbolurile **/*** reprezintă începutul comentariului, iar ***/** sfârșitul respectivului comentariu.

```
/* Acesta este un
comentariu care se
intinde pe mai multe randuri */
```

Nume

Prin **nume** dat unei variabile, clase, metode etc. înțelegem o succesiune de caractere care îndeplinește următoarele reguli:

- numele trebuie să înceapă cu o literă sau cu unul dintre caracterele "_" și "@";
- primul caracter poate fi urmat numai de litere, cifre sau un caracter de subliniere;
- numele care reprezintă cuvinte cheie nu pot fi folosite în alt scop decât acela pentru care au fost definite
- cuvintele cheie pot fi folosite în alt scop numai dacă sunt precedate de @
- două nume sunt distincte dacă diferă prin cel puțin un caracter (fie el și literă mică ce diferă de aceeași literă majusculă)

Convenții pentru nume:

- în cazul numelor claselor, metodelor, a proprietăților, enumerărilor, interfețelor, spațiilor de nume, fiecare cuvânt care compune numele începe cu majusculă
- în cazul numelor variabilelor dacă numele este compus din mai multe cuvinte, primul începe cu minusculă, celelalte cu majusculă

Cuvinte cheie în C#

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Simbolurile lexicale reprezentând constante, regulile de formare a expresiilor, separatorii de liste, delimitatorii de instrucțiuni, de blocuri de instrucțiuni, de șiruri de caractere etc. sunt în mare aceiași ca și în cazul limbajului C++.

3.6. Tipuri de date

În C# există două categorii de tipuri de date:

- **tipuri valoare**
  - tipul simple: **byte**, **char**, **int**, **float** etc.
  - tipul enumerare - **enum**
  - tipul structură - **struct**
- **tipuri referință**
  - tipul clasă - **class**
  - tipul interfață - **interface**
  - tipul delegat - **delegate**
  - tipul tablou - **array**

Toate tipurile de date sunt derivate din tipul **System.Object**

Toate **tipurile valoare** sunt derivate din clasa **System.ValueType**, derivată la rândul ei din clasa **Object** (alias pentru **System.Object**).

Limbajul C# conține un set de **tipuri predefinite** (int, bool etc.) și permite definirea unor tipuri proprii (enum, struct, class etc.).

Tipuri simple predefinite

Tip	Descriere	Domeniul de valori
object	rădăcina oricărui tip	
string	secvență de caractere Unicode	
sbyte	tip întreg cu semn, pe 8 biți	-128; 127
short	tip întreg cu semn, pe 16 biți	-32768; 32767
int	tip întreg cu semn pe, 32 biți	-2147483648; 21447483647
long	tip întreg cu semn, pe 64 de biți	-9223372036854775808; 9223372036854775807
byte	tip întreg fără semn, pe 8 biți	0; 255
ushort	tip întreg fără semn, pe 16 biți	0; 65535
uint	tip întreg fără semn, pe 32 biți	0; 4294967295
ulong	tip întreg fără semn, pe 64 biți	0; 18446744073709551615
float	tip cu virgulă mobilă, simplă precizie, pe 32 biți (8 pentru exponent, 24 pentru mantisă)	-3.402823E+38; 3.402823E+38
double	tip cu virgulă mobilă, dublă precizie, pe 64 biți (11 pentru exponent, 53 -mantisa)	-1.79769313486232E+308; 1.79769313486232E+308
bool	tip boolean	-79228162514264337593543950335; 79228162514264337593543950335
char	tip caracter din setul Unicode, pe 16 biți	
decimal	tip zecimal, pe 128 biți (96 pentru mantisă), 28 de cifre semnificative	

O valoare se asignează după următoarele reguli:

Sufix	Tip
nu are	**int, uint, long, ulong**
u, U	**uint, ulong**
L, L	**long, ulong**
ul, lu, Ul, lU, UL, LU, Lu	**ulong**

Exemple:

string s = "Salut!"	float g = 1.234F;
long a = 10;	double h = 1.234;
long b = 13L;	double i = 1.234D;
ulong c = 12;	bool cond1 = true;
ulong d = 15U;	bool cond2 = false;
ulong e = 16L;	decimal j = 1.234M;
ulong f = 17UL;	

Tipul enumerare

Tipul enumerare este un tip definit de utilizator. Acest tip permite utilizarea numelor care, sunt asociate unor valori numerice. Toate componentele enumerate au un același tip de bază întreg. În cazul în care, la declarare, nu se specifică tipul de bază al enumerării, atunci acesta este considerat implicit **int**.

Declararea unui tip enumerare este de forma:

```
enum [Nume_tip] [: Tip]o
{
    [identificator1][=valoare]o,
  ...
    [identificatorn][=valoare]o
}
```

Observații:

- În mod implicit valoarea primului membru al enumerării este 0, iar fiecare variabilă care urmează are valoarea (implicită) mai mare cu o unitate decât precedenta.
- Valorile folosite pentru inițializări trebuie să facă parte din domeniul de valori declarat al tipului
- Nu se admit referințe circulare:

```
enum ValoriCirculare
{
  a = b,
  b
}
```

Exemplu:

```
using System;
namespace tipulEnum
{
    class Program
    {
        enum lunaAnului
        {
            Ianuarie = 1,
            Februarie, Martie, Aprilie, Mai, Iunie, Iulie,
            August, Septembrie, Octombrie, Noiembrie, Decembrie
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Luna Mai este a {0}",
                          (int)lunaAnului.Mai + "-a luna din an.");
            Console.ReadLine();
        }
    }
}
```

În urma rulării programului se afișează mesajul:

`Luna Mai este a 5-a luna din an.`

Tablouri

Declararea unui tablou unidimensional:

`Tip[] nume;`

Prin aceasta, **nu** se alocă spațiu pentru memorare. Pentru a putea reține date în structura de tip tablou, este necesară o operație de **instanțiere**:

`nume = new Tip[NumarElemente];`

Declararea, instanțierea și chiar inițializarea tabloului se pot face în aceeași instrucțiune:

Exemplu:

`int[] v = new int[] {1,2,3};` sau

`int[] v = {1,2,3};  //new` este implicit

În cazul tablourilor cu mai multe dimensiuni facem distincție între **tablouri regulate** și **tablouri neregulate (tablouri de tablouri)**

Declarare în cazul **tablourilor regulate bidimensionale**:

`Tip[,] nume;`

Intanțiere:

`nume = new Tip[Linii,Coloane];`

Acces:

`nume[indice1,indice2]`

Exemple:
`int[,] mat = new int[,] {{1,2,3},{4,5,6},{7,8,9}};` sau
`int[,] mat = {{1,2,3},{4,5,6},{7,8,9}};`

Declarare în cazul **tablourilor neregulate bidimensionale**:

```
Tip[][] nume;
```

Intanțiere:

```
nume = new Tip[Linii],[];
nume[0]=new Tip[Coloane1]
...
```

`nume[Linii-1]=new Tip[Coloane`$_{Linii-1}$`]`

Acces:

```
nume[indice1][indice2]
```

Exemple:

```
int[][] mat = new int[][] {
                   new int[3] {1,2,3},
                   new int[2] {4,5},
                   new int[4] {7,8,9,1}
               };
```

sau

```
int[][] mat={new int[3] {1,2,3},new int[2] {4,5},new int[4] {7,8,9,1}};
```

Șiruri de caractere

Se definesc două tipuri de șiruri:

- regulate
- de tip „**verbatim**"

Tipul regulat conține între ghilimele zero sau mai multe caractere, inclusiv secvențe escape.

Secvențele escape permit reprezentarea caracterelor care nu au reprezentare grafică precum și reprezentarea unor caractere speciale: backslash, caracterul apostrof, etc.

Secvență escape	Efect
\'	apostrof
\"	ghilimele
\\	backslash
\0	null
\a	alarmă
\b	backspace
\f	form feed - pagină nouă
\n	new line - linie nouă
\r	carriage return - început de rând
\t	horizontal tab - tab orizontal
\u	caracter unicode
\v	vertical tab - tab vertical
\x	caracter hexazecimal

În cazul în care folosim multe secvențe escape, putem utiliza șirurile **verbatim**. Aceste șiruri pot să conțină orice fel de caractere, inclusiv caracterul EOLN. Ele se folosesc în special în cazul în care dorim să facem referiri la fișiere și la regiștri. Un astfel de șir începe întotdeauna cu simbolul**'@'** înaintea ghilimelelor de început.

Exemplu:

```
using System;
namespace SiruriDeCaractere
{
    class Program
    {
        static void Main(string[] args)
        {
            string a = "un sir de caractere";
            string b = "linia unu \nlinia doi";
            string c = @"linia unu
linia doi";
            string d="c:\\exemple\\unu.cs";
            string e = @"c:\exemple\unu.cs";
            Console.WriteLine(a); Console.WriteLine(b);
            Console.WriteLine(c); Console.WriteLine(d);
            Console.WriteLine(e); Console.ReadLine();
        }
    }
}
```

Programul va avea ieșirea

```
un sir de caractere
linia unu
linia doi
linia unu
linia doi
c:\exemple\unu.cs
c:\exemple\unu.cs
```

3.7. Conversii

3.7.1. Conversii numerice

În C# există două tipuri de conversii numerice:

- implicite
- explicite.

Conversia implicită se efectuează (automat) doar dacă nu este afectată valoarea convertită.

Exemplu:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Conversii
{
    class Program
    {
        static void Main(string[] args)
        {
            byte a = 13; // byte intreg fara semn
                         // pe 8 biti
            byte b = 20;
            long c;      //intreg cu semn pe 64 biti
            c = a + b;
            Console.WriteLine(c);
            Console.ReadLine();
        }
    }
}
```

Regulile de **conversie implicită** sunt descrise de tabelul următor:

din	în
sbyte	**short, int, long, float, double, decimal**
byte	**short, ushort, int, uint, long, ulong, float, double, decimal**
short	**int, long, float, double, decimal**
ushort	**int, uint, long, ulong, float, double, decimal**
int	**long, float, double, decimal**
uint	**long, ulong, float, double, decimal**
long	**float, double, decimal**
char	**ushort, int, uint, long, ulong, float, double, decimal**
float	**double**
ulong	**float, double, decimal**

Conversia explicită se realizează prin intermediul unei expresii cast, atunci când nu există posibilitatea unei conversii implicite.

Exemplu: în urma rulării programului

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Conversii1
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            int a = 5;
            int b = 2;
            float c;
            c = (float)a / b;  //operatorul cast
            Console.WriteLine("{0}/{1}={2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

se obține:
5/2 = 2,5

În cazul în care nu s-ar fi folosit operatorul **cast** rezultatul, evident eronat, ar fi fost:
5/2=2

Regulile de **conversie explicită** sunt descrise de tabelul următor:

din	în
sbyte	**byte, ushort, uint, ulong, char**
byte	**sbyte, char**
short	**sbyte, byte, ushort, uint, ulong, char**
ushort	**sbyte, byte, short, char**
int	**sbyte, byte, short, ushort, uint, ulong, char**
uint	**sbyte,byte, short, ushort, int, char**
long	**sbyte, byte, short, ushort, int, uint, ulong, char**
ulong	**sbyte, byte, short, ushort, int, uint, long, char**
char	**sbyte, byte, short**
float	**sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal**
double	**sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal**
decimal	**sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double**

3.7.2. Conversii între numere și șiruri de caractere

Limbajul C# oferă posibilitatea efectuării de conversii între numere și șiruri de caractere.

Sintaxa pentru conversia unui număr în șir de caractere:

număr → șir   "" + număr

sau se folosește metoda **ToString** a clasei **Object**.

Pentru conversia inversă, adică din șir de caractere în număr, sintaxa este:

șir → `int` `int.Parse(șir)` sau `Int32.Parse(șir)`
șir → `long` `long.Parse(șir` sau `Int64.Parse(șir)`
șir → `double` `double.Parse(șir)` sau `Double.Parse(șir)`
șir → `float` `float.Parse(șir)` sau `Float.Parse(șir)`

Observație: în cazul în care șirul de caractere nu reprezintă un număr valid, conversia acesui șir la număr va eșua.

Exemplu:

```
using System;

namespace Conversii
{
  class Program
  {
     static void Main(string[] args)
     {
       string s;
       const int a = 13;
       const long b = 100000;
       const float c = 2.15F;
       double d = 3.1415;
       Console.WriteLine("CONVERSII\n");
       Console.WriteLine("TIP\tVAL. \tSTRING");
       Console.WriteLine("———————————");
       s = "" + a;
       Console.WriteLine("int\t{0} \t{1}",a,s);
       s = "" + b;
       Console.WriteLine("long\t{0} \t{1}",b,s);
       s = "" + c;
       Console.WriteLine("float\t{0} \t{1}",c,s);
       s = "" + d;
       Console.WriteLine("double\t{0} \t{1}",d,s);
       Console.WriteLine("\nSTRING\tVAL \tTIP");
       Console.WriteLine("———————————");
       int a1;
       a1 = int.Parse("13");
       Console.WriteLine("{0}\t{1}\tint","13",a1);
       long b2;
       b2 = long.Parse("1000");
       Console.WriteLine("{0}\t{1}\tlong","1000",b2);
       float c2;
       c2 = float.Parse("2,15");
       Console.WriteLine("{0}\t{1}\tfloat","2,15",c2);
```

```
        double d2;
        d2 = double.Parse("3.1415",
                       System.Globalization.CultureInfo.InvariantCulture);
        Console.WriteLine("{0}\t{1}\tdouble","3.1415",d2);
        Console.ReadLine();
      }
   }
}
```

În urma rulării se obține:

```
CONVERSII
TIP     VAL.    STRING
————————————
int     13       13
long    100000  100000
float   2,15    2,15
double  3,1415   3,1415
STRING  VAL.    TIP
————————————
13       13      int
1000     1000    long
2,15     2,15    float
3.1415  3,1415  double
```

3.7.3. Conversii boxing și unboxing

Datorită faptului că în C# toate tipurile sunt derivate din clasa **Object** (**System.Object**), prin conversiile **boxing** (împachetare) și **unboxing** (despachetare) este permisă tratarea tipurilor valoare drept obiecte și reciproc. Prin conversia boxing a unui tip valoare, care se păstrează pe stivă, se produce ambalarea în interiorul unei instanțe de tip referință, care se păstrază în memoria heap, la clasa **Object**. Unboxing permite convertirea unui obiect într-un tipul valoare corespunzător.

Exemplu:

Prin boxing variabila **i** este asignata unui obiect **ob**:

```
int i = 13;
object ob = (object)i;  //boxing explicit
```

sau

```
int i = 13;
object ob = i;   //boxing implicit
```

Prin conversia de tip unboxing, obiectul **ob** poate fi asignat variabilei întregi **i**:

```
int i=13;
object ob = i;  //boxing implicit
i = (int)ob;    //unboxing explicit
```

3.8. Constante

În C# există două modalități de declarare a constantelor: folosind **const** sau folosind modificatorul **readonly**. Constantele declarate cu **const** trebuie să fie inițializate la declararea lor.

Exemple:

```
const int x;        //gresit, constanta nu a fost initializata
const int x = 13;   //corect
```

3.9. Variabile

O variabilă în C# poate să conțină fie o valoare a unui tip elementar, fie o referință la un obiect.

Exemple:

```
int Salut;
int  Azi_si _maine;
char caracter;
```

3.10. Expresii și operatori

Prin **expresie** se înțelege o secvență formată din **operatori** și **operanzi**. Un **operator** este un simbol ce indică acțiunea care se efectuează, iar **operandul** este valoarea asupra căreia se execută operația. În C# sunt definiți mai mulți operatori. În cazul în care într-o expresie nu intervin paranteze, operațiile se execută conform priorității operatorilor. În cazul în care sunt mai mulți operatori cu aceeași prioritate, evaluarea expresiei se realizează de la stânga la dreapta.

Prioritate	Tip	Operatori	Asociativitate
0	Primar	( ) [ ] f() . x++ x-- new typeof sizeof checked unchecked ->	→
1	Unar	+ - ! ~ ++x --x (tip) true false & sizeof	→
2	Multiplicativ	* / %	→
3	Aditiv	+ -	→
4	De deplasare	<< >>	→
5	Relațional	< > <= >= is as	→
6	De egalitate	== !=	→
7	AND (SI) logic	&	→
8	XOR (SAU exclusiv) logic	^	→
9	OR (SAU) logic	|	→
10	AND (SI) condițional	&&	→
11	OR (SAU) condițional	||	→
12	Condițional	?:	←
13	De atribuire	= *= /= %= += -= ^= &= <<= >>= |=	←

3.11. Instrucțiuni condiționale, de iterație și de control

Ne referim aici la instrucțiunile construite folosind cuvintele cheie: **if**, **else**, **do**, **while**, **switch**, **case**, **default**, **for**, **foreach**, **in**, **break**, **continue**, **goto**.

3.11.1 Instrucțiunea if

Instrucțiunea **if** are sintaxa:

```
if (conditie)
    Instructiuni_A;
else
    Instructiuni_B;
```

Exemplu:

```
using System;

namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            int n;
            Console.WriteLine("Introduceti un nr intreg ");
            n = Convert.ToInt32(Console.ReadLine());
            if (n >= 0)
                Console.WriteLine("Nr. introdus este > 0");
            else
                Console.WriteLine("Nr. introdus este < 0");
            Console.ReadLine();
        }
    }
}
```

3.11.2. Instrucțiunea while

Instrucțiunea **while** are sintaxa:

```
while (conditie) Instructiuni;
```

Cât timp **conditie** este indeplinită se execută **Instructiuni**.

Exemplu: Să se afișeze numerele întregi pozitive <= 10

```
using System;
```

```
namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            int n = 0;
            while (n <= 10)
            {
                Console.WriteLine("n este {0}", n);
                n++;
            }
            Console.ReadLine();
        }
    }
}
```

3.11.3. Instrucțiunea do – while

Instrucțiunea **do - while** are sintaxa este:

```
do
    Instructiuni;
while(conditie)
```

Exemplu: Asemănător cu exercițiul anterior, să se afișeze numerele întregi pozitive <= 10

```
using System;

namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            int n = 0;
            do
            {
                Console.WriteLine("n este {0}", n);
                n++;
            }
            while (n <= 10);
            Console.ReadLine();
        }
    }
}
```

3.11.4. Instrucțiunea for

Instrucțiunea **for** are sintaxa:

```
for (initializareCiclu; coditieFinal; pas)
    Instructiuni
```

Exemplu: Ne propunem, la fel ca în exemplele anterioare, să afișăm numerele pozitive <=10

```
using System;

namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            for(int n=0; n<=10; n++)
            {
                Console.WriteLine("n este {0}", n);
            }
            Console.ReadLine();
        }
    }
}
```

3.11.5. Instrucțiunea switch

La **switch** în C/C++, dacă la finalul intrucțiunilor dintr-o ramură **case** nu există **break**, se trece la următorul **case**. În C# se semnalează eroare. Există și aici posibilitatea de a face verificări multiple (în sensul de a trece la verificarea următoarei condiții din **case**) doar dacă **case**-ul nu conține instrucțiuni:

```
switch (a)
{
case 13:
case 20:
    x=5;
    y=8;
    break;
default:
    x=1;
    y-0;
    break;
}
```

Instrucțiunea **`switch`** admite în C# variabilă de tip șir de caractere care să fie comparată cu șirurile de caractere din **`case`**-uri.

Exemplu:

```
switch(strAnimal)
{
case "Pisica ":
    …
    break;
case "Catel ":
    …
    break;
default:
    …
    break;
}
```

3.11.6. Instrucțiunea foreach

Instrucțiunea **foreach** enumeră elementele dintr-o colecție, executând o instrucțiune pentru fiecare element. Elementul care se extrage este de tip read-only, neputând fi transmis ca parametru și nici aplicat un operator care să-l schimbe valoarea. Pentru a vedea cum acționează o vom compara cu instrucțiunea cunoscută **for**. Considerăm un vector nume format din șiruri de caractere:

```
string[] nume={"Ana", Ionel", "Maria"};
```

Să afișăm acest șir folosind instrucțiunea **for**:

```
for(int i=0; i<nume.Length; i++)
{
    Console.Write("{0} ", nume[i]);
}
```

Același rezultat îl obținem folosind instrucțiunea **foreach**:

```
foreach (string copil in nume)
{
    Console.Write("{0} ", copil);
}
```

3.11.7. Instrucțiunea break

Instrucțiunea **break** permite ieșirea din instrucțiunea cea mai apropiată **switch**, **while**, **do – while**, **for** sau **foreach**.

3.11.8. Instrucțiunea continue

Instrucțiunea **continue** permite reluarea iterației celei mai apropiate instrucțiuni **switch**, **while**, **do – while**, **for** sau **foreach**.

Exemplu:

```
using System;

namespace Salt
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            while(true)
            {
                Console.Write("{0} ",i);
                i++;
                if(i<10)
                    continue;
                else
                    break;
            }
            Console.ReadLine();
        }
    }
}
```

Se va afișa:

```
0 1 2 3 4 5 6 7 8 9
```

3.11.9. Instrucțiunea goto

Instrucțiunea **goto** poate fi folosită, în C#, în instrucțiunea **switch** pentru a face un salt la un anumit **case**.

Exemplu:

```
switch (a)
{
case 13:
    x=0;
    y=0;
    goto case 20;
case 15:
    x=3;
    y=1;
    goto default;
case 20:
    x=5;
    y=8;
    break;
default:
    x=1;
```

```
        y=0;
        break;
  }
```

3.12. Instrucțiunile try-catch-finally și throw

Prin *excepție* se înțelege un obiect care încapsulează informații despre situații anormale în funcționarea unui program. Ea se folosește pentru a semnala contextul în care apare o situație specială. De exemplu: erori la deschiderea unor fișiere, împărțire la 0 etc. Aceste erori se pot manipula astfel încât programul să nu se termine abrupt.

Sunt situații în care prefigurăm apariția unei erori într-o secvență de prelucrare și atunci integrăm secvența respectivă în blocul unei instrucțiuni **try**, precizând una sau mai multe secvențe de program pentru tratarea excepțiilor apărute (blocuri **catch**) și eventual o secvență comună care se execută după terminarea normală sau după "recuperarea" programului din starea de excepție (blocul **finally**).

Exemplu:

```
using System;
using System.IO;
namespace Exceptii
{
class tryCatch
{
    static void Main(string[] args)
    {
        Console.Write("Numele fisierului:");
        string s=Console.ReadLine(s);
        try
        {
            File.OpenRead(s);
        }
        catch (FileNotFoundException a)
        {
            Console.WriteLine(a.ToString());
        }
        catch (PathTooLongException b)
        {
            Console.WriteLine(b.ToString());
        }
        finally
        {
            Console.WriteLine("Programul s-a sfarsit");
            Console.ReadLine();
        }
    }
  }
}
```

Alteori putem simula prin program o stare de eroare ”aruncând” o excepție (instrucțiunea **`throw`**) sau putem profita de mecanismul de tratare a erorilor pentru a implementa un sistem de validare a datelor prin generarea unei excepții proprii pe care, de asemenea, o ”aruncăm” în momentul neîndeplinirii unor condiții puse asupra datelor.

Clasa **System.Exception** și derivate ale acesteia servesc la tratarea adecvată și diversificată a excepțiilor.

Exemplu: Considerăm clasele Copil, Fetita, Baiat definite fragmentat în capitolul 1. O posibilitate de validare la adăugara unui copil este aceea care generează o excepție proprie la depășirea dimensiunii vectorului static **`copii`**:

```
public static void adaug_copil(Copil c)
{
    if (nr_copii < nr_max)
        copii[nr_copii++] = c;
    else throw new Exception("Prea mulți copii");
}
```

CAPITOLUL 4

Programarea web cu ASP.NET

4.1. Introducere

ASP.NET este tehnologia Microsoft care permite dezvoltarea de aplicații web moderne, utilizând platforma Microsoft .NET cu toate beneficiile sale.

Pentru a înțelege procesul de realizare a unui site web cu ASP.NET este important să cunoaștem modul în care funcționează comunicarea între browser și serverul web. Acest proces este format din următoarele etape principale:

1 Browserul Web inițiaza o cerere (request) a unei resurse către serverul Web unde este instalată aplicația dorită.
2 Cererea este trimisă serverului Web folosind protocolul HTTP.
3 Serverul Web procesează cererea.
4 Serverul web trimite un răspuns browserului folosind protocolul HTTP.
5 Browserul procesează răspunsul în format HTML, afișând pagina web.
6 Utilizatorul poate introduce date (să spunem într-un formular), apasă butonul Submit și trimite date înapoi către server.
7 Serverul Web procesează datele.
8 Se reia de la pasul 4.

Serverul web primește cererea (request), iar apoi trimite un răspuns (response) înapoi către browser, după care conexiunea este închisă, și sunt eliberate resursele folosite pentru procesarea cererii. Acesta este modul de lucru folosit pentru afișarea paginilor statice (datele dintr-o pagină nu depind de alte date din alte pagini sau de alte acțiuni precedente ale utilizatorului) și nici o informație nu este stocată pe server. În cazul paginilor web dinamice, serverul poate să proceseze cereri de pagini ce conțin cod care se execută pe server, sau datele pot fi salvate pe server între două cereri din partea browserului.

Trimiterea datelor de la browser către server se poate realiza prin metoda GET sau POST.

Prin GET, URL-ul este completat cu un șir de caractere (QueryString) format din perechi de tipul cheie = valoare separate prin &.

Exemplu:

```
GET /getPerson.aspx?Id=1&city=Cluj HTTP/1.1
```

Folosind POST, datele sunt plasate în corpul mesajului trimis serverului:

Exemplu:

```
POST /getCustomer.aspx HTTP/1.1
Id=123&color=blue
```

Prin Get nu se pot trimite date de dimensiuni mari, iar datorită faptului că datele sunt scrise în URL-ul browser-ului, pot apărea probleme de securitate. De aceea, de preferat este să se folosească metoda POST pentru trimiterea de date.

Trimiterea datelor înapoi către server este numită deseori PostBack. Acțiunea de PostBack poate fi folosită atât cu metoda GET cât și cu metoda POST. Pentru a ști dacă se trimit date (POST) sau pagina este doar cerută de browser (GET), cu alte cuvinte pentru a ști dacă pagina curentă se încarcă pentru primă dată sau nu, în ASP.NET se folosește o proprietate a clasei Page numită IsPostBack.

4.2. Structura unei pagini ASP.NET

La crearea unui proiect nou, în fereastra Solution Explorer apare o nouă pagină web numită Default.aspx.

Orice pagină web .aspx este formată din 3 secțiuni: secțiunea de directive, secțiunea de cod, și secțiunea de layout.

Secțiunea de directive se folosește pentru a seta mediul de lucru, precizând modul în care este procesată pagina.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs" Inherits="_Default" %>
```

Secțiunea de cod, conține codul C# asociat paginii sau obiectelor din pagină. Codul poate fi plasat direct în pagina sau într-un fișier cu extensia *.cs*, cu același nume ca al paginii (de ex. *Default.aspx.cs*). În cazul în care se găsește direct în pagină, codul este cuprins între tag-urile <script> </script>:

```
<script runat="server">
    protected void Button1_Click(object sender, EventArgs e)
    {
Page.Title = "First Web Application";
    }
</script>
```

De obicei blocurile <script> conțin cod care se execută pe partea de client, însă dacă se folosește atributul runat = „server”, codul se va executa pe serverul web.

În cazul exemplului de mai sus, la apăsarea butonului se schimbă titlul paginii Web în browser.

În cazul în care în fereastra pentru adăugarea unei pagini noi în proiect, se bifează opțiunea *Place code in separate file*, codul este plasat într-un fișier separat, iar în secțiunea de directive este precizat numele acestui fișier.

Exemplu: `CodeFile="Default.aspx.cs"`.

Secțiunea de layout conține codul HTML din secțiunea Body:

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
           Text="Button" /></div>
    </form>
</body>
```

Atributul `runat="server"` pentru un anumit control, specifică faptul că pentru obiectul respectiv, ASP.NET Runtime Engine care rulează pe serverul web (IIS) va face transformarea într-un obiect HTML standard. Această conversie se realizează în funcție de tipul browserului, de varianta de javascript instalată pe browser și de codul C# asociat obiectului respectiv (numit *code behind*).

De exemplu pagina aspx de mai sus este transformată în următorul fișier html:

```
<form name="form1" method="post" action="Default.aspx" id="form1">
<div>
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMTQ20TkzNDMyMWRkIftHHP/CS/zQf/D4XczzogN1M1w=" />
</div>
<div> <br />
      <input type="submit" name="Button1" value="Button" id="Button1" style="z-
index: 102;left: 349px; position: absolute; top: 156px" />
</div>
<div>
<input type="hidden" name="__EVENTVALIDATION" = "__EVENTVALIDATION"   value =
"/wEWAgKlr8nLBAKM54rGBh7DPY7SctG1t7rMEnJSrO+1hHyP" />
</div>
</form>
```

Exemple complete de aplicații web puteți găsi pe DVD-ul Academic Resource Kit **(ARK)**, instalând **Resurse\Visual Studio 2005\101 Samples CS101SamplesAll.msi** sau descărcând cele 101 exemple de utilizare a Visual Studio 2005 de la adresa **http://msdn2.microsoft.com/en-us/vstudio/aa718334.aspx**. După instalare, din aplicația **Microsoft Visual Web Developer 2005** alegeți din meniul **File** opțiunea **Open Web Site** și selectați, din directorul **..\CS101SamplesAll\CS101SamplesWebDevelopment**, aplicația dorită. În fereastra **Solution Explorer** selectați **Start.aspx** și apoi butonul **View in Browser**".

4.3. Controale Server

Un control server poate fi programat, prin intermediul unui cod server-side, să răspundă la anumite evenimente din pagină. Își menține în mod automat starea între 2 cereri către server, trebuie să aibă atributul *id* și atributul *runat.*

Există două tipuri de controale server: Web și Html. Controalele server web oferă mai multe funcționalități programabile decât cele HTML. De asemenea pot detecta tipul browserului și pot fi transformate corespunzător în tag-urile html corespunzătoare. ASP.NET vine cu o suită foarte bogată de controale care pot fi utilizate de către programatori și care acoperă o foarte mare parte din funcționalitățile necesare unei aplicații web.

O proprietate importantă a controalelor server este *AutoPostBack.* Pentru a înțelege exemplificarea, vom considera o pagină în care avem un obiect de tip checkbox și un obiect de tip textbox care are proprietatea *visible = false.* În momentul în care este bifat checkbox-ul, vrem ca obiectul textbox să apară în pagină. Codul poate fi următorul:

```
protected void CheckBox1_CheckedChanged(object sender, EventArgs e)
{
    if (CheckBox1.Checked == true)
    {
        TextBox3.Visible = true;
        TextBox3.Focus();
    }
    else
    {
        TextBox3.Visible = false;
    }
}
```

Când vom rula pagina, vom constata că totuși nu se întâmplă nimic. Pentru a se executa metoda CheckBox1_CheckedCanged, pagina trebuie retrimisă serverului în momentul bifării checkbox-ului. Serverul trebuie să execute codul și apoi să retrimită către browser pagina în care textbox-ul este vizibil sau nu. De aceea controlul checkbox trebuie să genereze acțiunea de PostBack, lucru care se întâmplă dacă este setată valoarea true proprietății *AutoPostBack.* Unele controale generează întotdeauna Postback atunci când apare un anumit eveniment. De exemplu evenimentul *click* al controlului *button*.

Exemplu de folosire a controalelor web puteți găsi pe DVDul **ARK**, instalând **Resurse\Visual Studio 2005\101 Samples CS101SamplesAll.msi** sau descărcând cele 101 exemple de utilizare a Visual Studio 2005 de la adresa **http://msdn2.microsoft.com/en-us/vstudio/aa718334.aspx**, aplicația MenuAndSiteMapPath.

Pentru a înțelege mai bine fenomenul de PostBack, ne propunem să realizăm următoarea aplicație. Într-o pagină avem un textbox și un buton. Dorim ca în textbox să avem inițial (la încărcarea paginii) valoarea 0, și de fiecare dată când se apasă

butonul, valoarea din textbox să fie incrementată cu 1. Codul evenimentului Click al butonului și al evenimentului Load al paginii ar putea fi următorul:

```
    protected void Page_Load(object sender, EventArgs e)
    {
            TextBox1.Text = "0";
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
TextBox1.Text = Convert.ToString(Convert.ToInt32(TextBox1.Text) + 1) ;
    }
```

Vom observa, însă, că după prima incrementare valoarea în textbox rămâne 1. Acest lucru se întamplă deoarece evenimentul Load se execută la fiecare încărcare a paginii (indiferent că este vorba de request-ul inițial al browserului său de apelul de postback generat automat de evenimentul clic al butonului). Pentru a remedia această situație, obiectul Page în ASP are proprietarea *isPostBack,* a.î. putem să rescriem codul metodei Load:

```
protected void Page_Load(object sender, EventArgs e)
    {
        if (Page.IsPostBack == false) // nu este postback deci e prima
        {                                // incarcare a paginii
            TextBox1.Text = "0";
        }
    }
```

4.4. Păstrarea informațiilor în aplicațiile web

Există o deosebire fundamentală între aplicațiile Windows și cele Web. Anume, în aplicațiile Windows odată creat un obiect acesta rămâne în memorie în principiu până la terminarea aplicației și va putea fi utilizat și din alte ferestre decât cele în care a fost creat, atâta timp cât este public. Pe de altă parte, în aplicațiile web paginile nu se păstrează în memorie pe calculatorul utilizatorului (clientului) iar aici ne vom pune problema păstrării informațiilor.

Când browserul cere o anumită pagină, ea este încărcată de serverul web, se execută codul asociat pe baza datelor trimise de user, rezultând un răspuns în format html trimis browserului. După ce este prelucrată pagina de către server, obiectele din pagină sunt șterse din memorie, pierzând astfel valorile. De aceea apare întrebarea: cum se salvează/transmit informațiile între paginile unui site web sau chiar în cadrul aceleiași pagini, între două cereri succesive către server?

4.4.1. Păstrarea stării controalelor

Obiectul ViewState

Starea controalelor unei pagini este pastrată automat de către ASP.NET și astfel nu trebuie să ne facem griji cu privire la informațiile care apar în controale pentru ca

ele nu vor dispărea la următorul PostBack — adică la următoarea încărcare a paginii curente. De exemplu, dacă scriem un text într-o căsuță de text și apoi apăsăm un buton care generează un PostBack iar pagina se reîncarcă, ea va conține căsuța de text respectivă cu textul introdus de noi înainte de reîncărcare.

În momentul generării codului Html de către server se generează un control html de tip <input type=”hidden”...> , a cărui valoare este un șir de caractere ce codifică starea controalelor din pagină:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwULLTE1OTg1NDYyNDZkZFCFstl/DwSGv81TuCB397Tk5+CJ" />
```

Se pot adăuga valori în ViewState și de către programator, folosind obiectul ViewState cu metoda Add (cheie, valoare_obiect): `ViewState.Add("TestVariable", "Hello");`

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Page.IsPostBack == false)
    {
        ViewState.Add("ViewStateTest", "Hello");
    }
}
```

Regăsirea datelor se realizează folosind ca indice numele obiectului:

```
protected void Button1_Click(object sender, EventArgs e)
{
    TextBox1.Text = ViewState["ViewStateTest"].ToString();
}
```

4.4.2. Păstrarea altor informații

Așa cum am observat în paragraful anterior, starea controalelor de pe o anumită pagină web ASP.NET se pastrează între mai multe cereri către server pentru aceeași pagină, folosind obiectul ViewState în mod automat, transparent pentru programator.

Dacă dorim să păstrăm mai multe informații decât doar conținutul controalelor, cum ar fi valorile unor variabile instanțiate într-o anumită pagină, atunci va trebui să o facem explicit, pentru că acestea se pierd în momentul în care serverul web regenerează pagina curentă, ceea ce se întâmplă la fiecare *PostBack* , cum se întâmplă de exemplu la apăsarea unui buton ASP.NET.

4.4.2.1. Profile

O posibilitate de păstrare a informațiilor specifice unui utilizator constă în folosirea obiectului Profile, prin intermediul fișierului de configurare Web.Config. Acesta este un fișier XML în care se rețin opțiuni de configurare. Pentru a adăuga o proprietate obiectului profile, în fișierul Web.Config se adaugă:

```
        <profile enabled="true">
          <properties>
            <add  name ="ProfileTest" allowAnonymous ="true"/>
          </properties>
   </profile>
```

Atributul name reține numele proprietății. După aceste modificări, proprietatea definită în Web.config poate fi apelată pentru obiectul Profile:

```
Profile.ProfileTest = "Hello world";
Sau
Label1.Text = Profile.ProfileTest;
```

4.4.2.2. Session

Obiectul Session este creat pe serverul web la prima accesare a sitului de către un utilizator și rămâne în memorie în principiu atât timp cât utilizatorul rămâne conectat la site. Există și excepții, dar ele nu fac obiectul acestui material.

Pentru a adăuga un obiect în sesiune, trebuie doar să scriem un cod de genul următor:

```
protected void Button1_Click(object sender, EventArgs e)
{
    Session["sir"] = test;
}
```

Session este de fapt un dicționar (listă de perechi cheie — valoare), în care valorile sunt de tip *object*. Ceea ce înseamnă că la citirea unor valori din sesiune va trebui să realizăm o conversie de tip.

```
protected void Button2_Click(object sender, EventArgs e)
{
    test = Session["sir"].ToString();
    TextBox1.Text = test;
}
```

Odată introdus un obiect în Session, el poate fi accesat din toate paginile aplicației, atât timp cât el există acolo. Programatorul poate realiza scoaterea obiectului din sesiune atunci când dorește acest lucru:

```
Session.Remove("sir");
```

4.4.2.3. Application

Obiectul Application se comportă în mod identic cu Session, doar că este specific întregii aplicații, adică tuturor utilizatorilor care acceseaza un site web la un moment

dat, și nu unei anumite sesiuni. Cu alte cuvinte odată introdus un obiect în Applicatio, va putea fi accesat din orice loc al sitului și de către toți utilizatorii acestuia.

4.4.2.4. Membrii statici

Toate variabilele declarate ca fiind statice sunt specifice întregii aplicații și nu unei anumite sesiuni. De exemplu, dacă atunci când un site este accesat de Utilizator1 și o variabilă declarată:

```
static string test = "init";
```

se modifică de către acesta:

```
test = "modificat";
```

atunci toți utilizatorii aplicației vor vedea valoarea modificată din acel moment înainte.

4.4.3. Concluzii

În cazul obiectului ViewState, datele sunt salvate în pagina web sub forma unui șir de caractere, iar în cazul obiectului Session respectiv Application în memoria serverului web. Dacă datele salvate sunt de dimensiuni mari, în primul caz crește dimensiunea paginii web, care va fi transmisă mai încet, iar în al doilea caz rezultă o folosire excesivă a memoriei serverului web, ceea ce duce la scăderea vitezei de lucru. Această folosire excesivă a memoriei poate să apară și în cazul unei dimensiuni a datelor ceva mai redusă, dar a unui număr mare de utilizatori care accesează simultan pagina (pentru fiecare se va creea un obiect sesiune).

4.5. Validarea datelor

În toate aplicațiile web și nu numai se pune problema validării datelor introduse de utilizator. Cu alte cuvinte, trebuie să ne asigurăm că utilizatorul site-ului nostru introduce numai date corecte în căsuțele de text care îi sunt puse la dispoziție. De exemplu, dacă pe o pagină web se cere utilizatorului introducerea vârstei sale și pentru asta îi punem la dispozitie o căsuță de text, va fi obligatoriu să ne asigurăm că în acea căsuță se pot introduce numai cifre și că numărul rezultat este încadrat într-un anumit interval. Sau, un alt exemplu, este introducerea unei adrese de email validă din punct de vedere al formatului.

ASP.NET vine cu o serie de controale gata create în scopul validării datelor. Aceste controale sunt de fapt clase care provin din aceeași ierarhie, având la bază o clasă cu proprietăți comune tuturor validatoarelor.

4.5.1. Proprietăți comune

1 **ControlToValidate**: este proprietatea unui control de validare care arată spre controlul (căsuța de text) care trebuie să fie validat.

2 **ErrorMessage**: reprezintă textul care este afișat în pagina atunci când datele din

controlul de validat nu corespund regulii alese.

3 **EnableClientSideScript**: este o proprietate booleană care specifică locul în care se execută codul de validare (pe client sau pe server).

4 **Alte proprietăți, specifice tipului de validator.**

4.5.2. Validatoare

1 **RequiredFieldValidator.** Verifică dacă în căsuța de text asociată prin proprietatea ControlToValidate s-a introdus text. Util pentru formularele în care anumite date sunt obligatorii.

2 **RangeValidator**. Verifică dacă informația introdusă în căsuța de text asociată face parte dintr-un anumit interval, specificat prin tipul datei introduse (prorietatea **Type**) și **MinimumValue** respectiv **MaximumValue**.

3 **RegularExpressionValidator.** Verifică dacă informația din căsuța de text asociată este conform unei expresii regulate specificate. Este util pentru validarea unor informații de genul adreselor de email, numerelor de telefon, etc — în general informații care trebuie să respecte un anumit format. Trebuie setată proprietatea **ValidationExpression** în care se pot alege câteva expresii uzuale gata definite.

4 **CompareValidator.** Compară datele introduse în căsuța de text asociată cu o valoare prestabilită (**ValueToCompare**), în funcție de operatorul ales (proprietatea **Operator**) și de tipul de date care se așteaptă (proprietatea **Type**).

Pe lângă validatoarele prezentate mai sus, programatorul poate crea validatoare customizate, care să verifice datele introduse de utilizator conform unor reguli proprii.

Exemplu de folosire a validărilor pentru un modul de login puteți găsi pe dvd-ul **ARK**, instalând **Resurse\Visual Studio 2005\101 Samples CS101SamplesAll.msi** sau descărcând cele 101 exemple de utilizare a Visual Studio 2005 de la adresa **http://msdn2.microsoft.com/en-us/vstudio/aa718334.aspx**, aplicația Membership.

4.6. Securitatea în ASP.NET

Pentru o aplicație securizată, avem mai multe posibilități de autentificare, cele mai des întâlnite fiind sintetizate în tabelul de pe slide. Implementarea politicii de securitate se poate face atât din IIS cât și din aplicația ASP.NET.

Tipul aplicației	Modul de autentificare	Descriere
Aplicație web publică pe Internet.	Anonim	Nu avem nevoie de securizare.
Aplicație web pentru Intranet.	Windows Integrated	Acest mod autentifică utilizatorii folosind lista de useri de pe server (Domain Controller). Drepturile userilor în aplicația web este dat de nivelul de privilegii al contului respectiv.
Aplicație web disponibilă pe Internet, dar cu acces privat.	Windows Integrated	Utilizatorii companiei pot accesa aplicația din afara Intranetului, folosind conturi din lista serverului (Domain Controller).
Aplicații web comerciale.	Forms Authentication	Aplicații care au nevoie de informații confidențiale și eventual în care sunt mai multe tipuri de utilizatori.

4.6.1. Windows Authentication

În acest mod de autentificare, aplicația ASP .NET are încorporate procedurile de autentificare, dar se bazează pe sistemul de operare Windows pentru autentificarea utilizatorului.

1. Utilizatorul solicită o pagină securizată de la aplicația Web.
2. Cererea ajunge la Serverul Web IIS care compară datele de autentificare ale utilizatorului cu cele ale aplicației (sau ale domeniului)
3. Dacă acestea două nu corespund, IIS refuză cererea utilizatorului
4. Calculatorul clientului generează o fereastră de autentificare
5. Clientul introduce datele de autentificare, după care retrimite cererea către IIS
6. IIS verifică datele de autentificare, și în cazul în care sunt corecte, direcționează cererea către aplicația Web.
7. Pagina securizată este returnată utilizatorului.

4.6.2. Forms-Based Authentication

Atunci când se utilizează autentificarea bazată pe formulare, IIS nu realizează autentificarea, deci este necesar ca în setările acestuia să fie permis accesul anonim.

1. În momentul în care un utilizator solicită o pagină securizată, IIS autentifică clientul ca fiind un utilizator anonim, după care trimite cererea către ASP.NET
2. Acesta verifică pe calculatorul clientului prezența unui anumit cookie1
3. Dacă cookie-ul nu este prezent sau este invalid, ASP.NET refuză cererea clientului și returnează o pagină de autentificare (Login.aspx)
4. Clientul completează informațiile cerute în pagina de autentificare și apoi trimite informațiile
5. Din nou, IIS autentifică clientul ca fiind un utilizator anonim și trimite cererea către ASP.NET
6. ASP.NET autentifică clientul pe baza informațiilor furnizate. De asemenea generează și un cookie. Cookie reprezintă un mic fișier text ce păstrează diverse informații despre utilizatorul respectiv, informații folosite la următoarea vizită a sa pe site-ul respectiv, la autentificare, sau în diverse alte scopuri.
7. Pagina securizată cerută și noul cookie sunt returnate clientului. Atâta timp cât acest cookie rămâne valid, clientul poate solicita și vizualiza orice pagină securizată ce utilizează aceleași informații de autentificare.

4.6.3. Securizarea unei aplicații web

Securizarea unei aplicații web presupune realizarea a două obiective: (1) autentificarea și (2) autorizarea.

1. **Autentificarea** presupune introducerea de către utilizator a unor credențiale, de exemplu nume de utilizator și parolă, iar apoi verificarea în sistem că acestea există si sunt valide.

2. **Autorizarea** este procesul prin care un utilizator autentificat primește acces pe resursele pe care are dreptul să le acceseze.

Aceste obiective pot fi atinse foarte ușor utilizând funcționalitățile și uneltele din ASP.NET respectiv Visual Studio, anume clasa *Membership* și unealta *ASP.NET Configuration* (din meniul Website al Visual Studio Web Developer Express). Configurarea autentificării și autorizării se poate realiza după cum se vede în acest tutorial:

http://msdn2.microsoft.com/en-us/library/879kf95c(VS.80).aspx.

Un exemplu de securizare a aplicaților web puteți găsi pe dvd-ul **ARK**, instalând **Resurse\Visual Studio 2005\101 Samples CS101SamplesAll.msi** sau descărcând cele 101 exemple de utilizare a Visual Studio 2005 de la adresa **http://msdn2.microsoft.com/en-us/vstudio/aa718334.aspx**, aplicatia Security.

4.7. Accesul la o baza de date într-o pagină web

Pentru adăugarea unei baze de date proiect, din meniul *Add Item* se alege *SQL Database*. Baza de date va fi adaugată în directorul *App_data* al proiectului.

Legătura între baza de date și controalele html se realizează prin intermediul obiectului SqlDataSource. Din meniul contextual asociat acestui obiect se alege opțiunea *Configure Data Source,* se alege baza de date, și se construiește interogarea SQL pentru regăsirea datelor.

La această sursă de date se pot lega controale de afișare a datelor cum ar fi: *GridView, Detailview, FormView.* Din meniul contextual asociat acestor controale se alege opțiunea *Choose data source*, de unde se alege sursa de date.

Un exemplu de acces la o bază de date într-o aplicație web puteți găsi pe DVD-ul **ARK** instalând **Resurse\Visual Studio 2005\101 Samples CS101SamplesAll.msi** sau descărcând cele 101 exemple de utilizare a Visual Studio 2005 de la adresa http://msdn2.microsoft.com/en-us/vstudio/aa718334.aspx, aplicația DataControls. Pentru acest exemplu va trebui să descărcați baza de date AdventureWorksDB de la adresa **http://www.codeplex.com/MSFTDBProdSamples/Release/ProjectReleases.aspx?ReleaseId=4004**. Fișierul descărcat va fi unul de tip *.msi care trebuie lansat pentru a instala baza de date pe server/calculator.

4.8. Resurse

Dezvoltarea de aplicații web cu Visual Web Developer Express:
http://msdn.microsoft.com/vstudio/express/vwd/

CAPITOLUL 5

Programare vizuală

5.1. Concepte de bază ale programării vizuale

Programarea vizuală trebuie privită ca un mod de proiectare a unui program prin operare directă asupra unui set de elemente grafice (de aici vine denumirea de programare vizuală). Această operare are ca efect scrierea automată a unor secvențe de program, secvențe care, împreună cu secvențele scrise textual[28], vor forma programul.

Spunem că o *aplicație* este *vizuală* dacă dispune de o interfață grafică sugestivă și pune la dispoziția utilizatorului instrumente specifice de utilizare (*drag*, *clic*, *hint* etc.)

Realizarea unei aplicații vizuale nu constă doar în desenare și aranjare de controale, ci presupune în principal stabilirea unor decizii arhitecturale[29], decizii ce au la bază unul dintre **modelele arhitecturale** de bază:

a) Modelul arhitectural **orientat pe date.**
Acest model nu este orientat pe obiecte, timpul de dezvoltare al unei astfel de aplicații este foarte mic, o parte a codului este generată automat de Visual Stdio.Net, codul nu este foarte ușor de întreținut și este recomandat pentru aplicații relativ mici sau cu multe operații de acces (in/out) la o bază de date.

b) Modelul arhitectural **Model-view-controller**
Este caracterizat de cele trei concepte de bază: **Model (**reprezentarea datelor se realizează într-o manieră specifică aplicației: conține obiectele de „business”, încapsulează accesul la date), **View** (sunt utilizate elemente de interfață, este format din Form-uri), **Controller**(procesează și răspunde la evenimente iar SO, clasele Form și Control din .Net rutează evenimentul către un „handler”, eveniment tratat în codul din spatele Form-urilor).

c) Modelul arhitectural **Multi-nivel**

Nivelul de prezentare (interfața)

Se ocupă numai de afișarea informațiilor către utilizator și captarea celor introduse de acesta. Nu cuprinde detalii despre logica aplicației, și cu atât mai mult despre baza de date

[28] Se utilizează ades antonimia dintre vizual (operații asupra unor componente grafice) și textual (scriere de linii de cod); proiectarea oricărei aplicații "vizuale" îmbină ambele tehnici.

[29] Deciziile arhitecturale stabilesc în principal cum se leagă interfața de restul aplicației și cât de ușor de întreținut este codul rezultat.

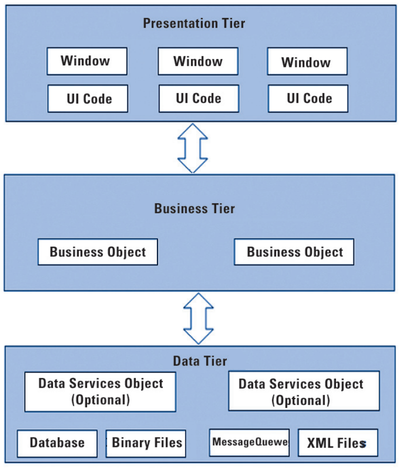
sau fișierele pe care aceasta le utilizează. Cu alte cuvinte, în cadrul interfeței cu utilizatorul, nu se vor folosi obiecte de tipuri definite de programator, ci numai baza din .NET.

Nivelul de logică a aplicației

Se ocupă de tot ceea ce este specific aplicației care se dezvoltă. Aici se efectuează calculele și procesările și se lucrează cu obiecte de tipuri definite de programator.

Nivelul de acces la date

Aici rezidă codul care se ocupă cu accesul la baza de date, la fișiere, la alte servicii.



Această ultimă structură este foarte bună pentru a organiza aplicațiile, dar nu este ușor de realizat. De exemplu, dacă în interfața cu utilizatorul prezentăm date sub formă *ListView* și la un moment dat clientul ne cere reprezentarea datelor într-un *GridView*, modificările la nivel de cod nu se pot localiza doar în interfață deoarece cele două controale au nevoie de modele de acces la date total diferite.

Indiferent de modelul arhitectural ales, în realizarea aplicației mai trebuie respectate și **principiile proiectării interfețelor**:

Simplitatea

Interfața trebuie să fie cât mai ușor de înțeles[30] și de învățat de către utilizator și să permită acestuia să efectueze operațiile dorite în timp cât mai scurt. În acest sens, este vitală culegerea de informații despre utilizatorii finali ai aplicației și a modului în care aceștia sunt obișnuiți să lucreze.

Poziția controalelor

Locația controalelor dintr-o fereastră trebuie să reflecte importanța relativă și frecvența de utilizare. Astfel, când un utilizator trebuie să introducă niște informații — unele obligatorii și altele opționale — este indicat să organizăm controalele astfel încât primele să fie cele care preiau informații obligatorii.

Consistența

Ferestrele și controalele trebuie să fie afișate după un design asemănător („tem-

[30] Întrucât mintea umană poate să perceapă la un moment dat aproximativ 5-9 obiecte, o fereastră supra-încărcată de controale o face greu de utilizat..

plate”) pe parcursul utilizării aplicației. Înainte de a implementa interfața, trebuie decidem cum va arăta aceasta, să definim „template”-ul.

Estetica

Intefața trebuie să fie pe cât posibil plăcută și atrăgătoare.

5.2. Mediul de dezvoltare Visual C#

Mediul de dezvoltare **Microsoft Visual C#** dispune de instrumente specializate de proiectare, ceea ce permite crearea aplicațiilor în mod interactiv, rapid și ușor.

Pentru a construi o aplicație Windows (File→New Project) se selectează ca template *Windows Application*.

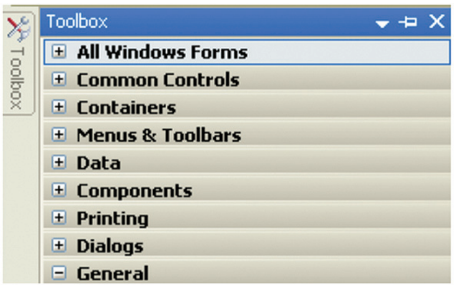
O aplicație Windows conține cel puțin o fereastră (*Form*) în care se poate crea o interfață cu utilizatorul aplicației.

Componentele vizuale ale aplicației pot fi prelucrate în modul **Designer** (**Shift+F7**) pentru a plasa noi obiecte, a le stabili proprietățile etc. Codul ”din spatele” unei componente vizuale este accesibil în modul **Code** (**F7**).

În fereastra **Solution Explorer** sunt afișate toate fișierele pe care Visual Studio.NET le-a inclus în proiect. **Form1.cs** este formularul creat implicit de Visual Studio.NET ca parte a proiectului.

Fereastra **Properties** este utilizată pentru a schimba proprietățile obiectelor.

Toolbox conține **controale standard** drag-and-drop și componente utilizate în crearea aplicației Windows. Controalele sunt grupate în categoriile logice din imaginea alăturată.



Designer, **Code**, **Solution Explorer** și celelalte se află grupate în meniul **View**.

La crearea unei noi aplicații vizuale, Visual Studio.NET generează un spațiu de nume ce conține clasa statică *Program*, cu metoda statică ce constituie punctul de intrare (de lansare) a aplicației:

```
static void Main()
        {   ...
            Application.Run(new Form1());
        }
```

Clasa **Application** este responsabilă cu administrarea unei aplicații Windows, punând la dispoziție proprietăți pentru a obține informații despre aplicație, metode de lucru cu aplicația și altele. Toate metodele și proprietățile clasei Application sunt

statice. Metoda Run invocată mai sus creează un formular implicit, aplicația răspunzând la mesajele utilizatorului până când formularul va fi închis.

Compilarea modulelor aplicației și asamblarea lor într-un singur fișier ”executabil” se realizează cu ajutorul opțiunilor din meniul Build, uzuală fiind **Build Solution** (**F6**).

Odată implementată, aplicația poate fi lansată, cu asistență de depanare sau nu (opțiunile **Start** din meniul *Debug*). Alte facilități de depanare pot fi folosite prin umărirea pas cu pas, urmărirea până la puncte de întrerupere etc. (celelalte opțiuni ale meniului *Debug*). Ferestre auxiliare de urmărire sunt vizualizate automat în timpul procesului de depanare, sau pot fi activate din submeniul *Windows* al meniului *Debug*.

5.3. Ferestre

Spațiul Forms ne oferă clase specializate pentru: creare de ferestre sau *formulare* (**System.Windows.Forms.Form**), elemente specifice (*controale*) cum ar fi butoane (**System.Windows.Forms.Button**), casete de text (**System.Windows.Forms.TextBox**) etc. Proiectarea unei ferestre are la bază un cod complex, generat automat pe măsură ce noi desemnăm componentele și comportamentul acesteia. În fapt, acest cod realizează: derivarea unei clase proprii din **System.Windows.Forms.Form,** clasă care este înzestrată cu o *colecție de controale* (inițial vidă). Constructorul ferestrei realizează instanțieri ale claselor *Button*, *MenuStrip*, *Timer* etc. (orice plasăm noi în fereastră) și adaugă referințele acestor obiecte la colecția de controale ale ferestrei.

Dacă modelul de fereastră reprezintă ferestra principală a aplicației, atunci ea este instanțiată automat în programul principal (metoda *Main*). Dacă nu, trebuie să scriem noi codul ce realizează instanțierea.

Clasele derivate din Form moștenesc o serie de proprietăți care determină atributele vizuale ale ferestrei (stilul marginilor, culoare de fundal, etc.), metode care implementează anumite comportamente (*Show*, *Hide*, *Focus* etc.) și o serie de metode specifice (*handlere*) de tratare a evenimentelor (*Load, Clic etc.).*

O fereastră poate fi activată cu **form.Show()** sau cu **form.ShowDialog()**, metoda a doua permițând ca revenirea în fereastra din care a fost activat noul formular să se facă numai după ce noul formular a fost inchis (spunem că formularul nou este deschis **modal**).

Un **propietar** este o fereastră care contribuie la comportarea formularului deținut. Activarea propietarului unui formular deschis modal va determina activarea formularului deschis modal. Când un nou formular este activat folosind `form.Show()` nu va avea nici un deținător, acesta stabilindu-se direct:

```
public Form Owner { get; set; }
F_nou form=new F_nou();
form.Owner = this; form.Show();
```

Formularul deschis modal va avea un proprietar setat pe **null.** Deținătorul se poate stabili setând proprietarul înainte să apelăm **Form.ShowDialog()** sau apelând

From.ShowDialog() cu proprietarul ca argument.

```
F_nou form = new F_nou();form.ShowDialog(this);
```

Vizibilitatea unui formular poate fi setată folosind metodele `Hide` sau `Show`. Pentru a ascunde un formular putem folosi:

```
this.Hide();            // setarea propietatii Visible indirect sau
this.Visible = false; // setarea propietatii Visible direct
```

Printre cele mai uzuale proprietăți ale form-urilor, reamintim:

- **StartPosition** determină poziția ferestrei atunci când aceasta apare prima dată, poziție ce poate fi setată **Manual** sau poate fi centrată pe desktop (**CenterScreen**), stabilită de Windows, formularul având dimensiunile și locația stabilite de programator (**WindowsDefaultLocation**) sau Windows-ul va stabili dimensiunea inițială și locația pentru formular (**WindowsDefaultBounds**) sau, centrat pe formularul care l-a afișat (**CenterParent**) atunci când formularul va fi afișat modal.
- **Location (X,Y)** reprezintă coordonatele colțului din stânga sus al formularului relativ la colțul stânga sus al containerului. (Această propietate e ignorată dacă StartPosition = Manual). Mișcarea formularului (și implicit schimbarea locației) poate fi tratată în evenimentele **Move** și **LocationChanged** .
  Locația formularului poate fi stabilită relativ la desktop astfel:

  ```
  void Form_Load(object sender, EventArgs e) {
  this.Location = new Point(1, 1);
  this.DesktopLocation = new Point(1, 1); } //formularul in desktop
  ```

- **Size** (**Width** și **Height**) reprezintă dimensiunea ferestrei. Când se schimbă proprietățile Width și Height ale unui formular, acesta se va redimensiona automat, această redimensionare fiind tratată în evenimentele **Resize** sau in **SizeChanged**. Chiar dacă propietatea **Size** a formularului indică dimensiunea ferestrei, formularul nu este în totalitate responsabil pentru desenarea întregului conținut al său. Partea care este desenată de formular mai este denumită și Client Area. Marginile, titlul și scrollbar-ul sunt desenate de Windows.
- **MaxinumSize** și **MinimumSize** sunt utilizate pentru a restricționa dimensiunile unui formular.

  ```
  void Form_Load(object sender, EventArgs e) {
  this.MinimumSize = new Size(200, 100);...
  this.MaximumSize = new Size(int.MaxValue, 100);...}
  ```

- **IsMdiContainer** precizează dacă form-ul reprezintă un container pentru alte form-uri.
- **ControlBox** precizează dacă fereastra conține sau nu un icon, butonul de închidere

al ferestrei și meniul System (Restore, Move, Size, Maximize, Minimize, Close).

- **HelpButton**-precizează dacă butonul va apărea sau nu lângă butonul de închidere al formularului (doar dacă MaximizeBox=false, MinimizeBox=false). Dacă utilizatorul apasă acest buton și apoi apasă oriunde pe formular va apărea evenimentul **HelpRequested** (F1).
- **Icon** reprezintă un obiect de tip *.ico folosit ca icon pentru formular.
- **MaximizeBox** și **MinimizeBox** precizează dacă fereastra are sau nu butonul Maximize și respectiv Minimize
- **Opacity** indică procentul de opacitate[31]
- **ShowInTaskbar** precizează dacă fereastra apare in TaskBar atunci când formularul este minimizat.
- **SizeGripStyle** specifică tipul pentru 'Size Grip' (Auto, Show, Hide). Size grip (în colțul din dreapta jos) indică faptul că această fereastră poate fi redimensionată.
- **TopMost** precizează dacă fereastra este afisată în fața tuturor celorlalte ferestre.
- **TransparencyKey** identifică o culoare care va deveni transparentă pe formă.

Definirea unei funcții de tratare a unui **eveniment** asociat controlului se realizează prin selectarea grupului *Events* din ferestra *Properties* a controlului respectiv și alegerea evenimentului dorit.

Dacă nu scriem nici un nume pentru funcția de tratare, ci efectuăm dublu clic în căsuța respectivă, se generează automat un nume pentru această funcție, ținând cont de numele controlului și de numele evenimentului (de exemplu `button1_Click`).

Dacă în **Designer** efectuăm dublu clic pe un control, se va genera automat o funcție de tratare pentru evenimentul implicit asociat controlului (pentru un buton evenimentul implicit este *Click*, pentru *TextBox* este *TextChanged*, pentru un formular *Load* etc.).

Printre **evenimentele** cele mai des utilizate, se numără :

- **Load** apare când formularul este pentru prima data încărcat în memorie.
- **FormClosed** apare când formularul este închis.
- **FormClosing** apare când formularul se va inchide ca rezultat al acțiunii utilizatorului asupra butonului Close (Dacă se setează CancelEventArgs.Cancel =True atunci se va opri închiderea formularului).
- **Activated** apare pentru formularul activ.
- **Deactivate** apare atunci când utilizatorul va da clic pe alt formular al aplicatiei.

5.4. Controale

Unitatea de bază a unei interfețe Windows o reprezintă un control. Acesta poate fi „găzduit” de un container ce poate fi un formular sau un alt control.

Un control este o instanță a unei clase derivate din System.Windows.Forms și este reponsabil cu desenarea unei părți din container. Visual Studio .NET vine cu o serie de controale standard, disponibile în Toolbox. Aceste controale pot fi grupate astfel:

[31] Dacă va fi setată la 10% formularul și toate controalele sale vor fi aproape invizibile.

5.4.1. Controale form. Controlul form este un container. Scopul său este de a găzdui alte controale. Folosind proprietățile, metodele și evenimentele unui formular, putem personaliza programul nostru.

În tabelul de mai jos veți găsi o listă cu controalele cel mai des folosite și cu descrierea lor. Exemple de folosire a acestor controale vor urma după explicarea proprietăților comune al controalelor și formularelor.

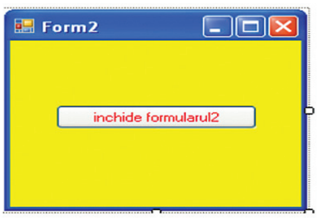
Funcția controlului	Numele controlului	Descriere
buton	**Button**	Sunt folosite pentru a executa o secvență de instrucțiuni în momentul activării lor de către utilizator
calendar	**MonthCalendar**	Afișează implicit un mic calendar al lunii curente. Acesta poate fi derulat și înainte și înapoi la celelalte luni calendaristice.
casetă de validare	**CheckBox**	Oferă utilizatorului opțiunile : da/nu sau include/exclude
etichetă	**Label**	Sunt folosite pentru afișarea etichetelor de text, și a pentru a eticheta controalele.
casetă cu listă	**ListBox**	Afișează o listă de articole din care utilizatorul poate alege.
imagine	**PictureBox**	Este folosit pentru adăugarea imaginilor sau a altor resurse de tip bitmap.
pointer	**Pointer**	Este utilizat pentru selectarea, mutarea sau redimensionarea unui control.
buton radio	**RadioButton**	Este folosit pentru ca utilizatorul să selecteze un singur element dint-un grup de selecții.
casetă de text	**TextBox**	Este utilizat pentru afișarea textului generat de o aplicație sau pentru a primi datele introduse de la tastatură de către utilizator.

5.4.2. Proprietăți comune ale controalelor și formularelor:

Proprietatea Text Această proprietate poate fi setată în timpul proiectării din fereastra Properties, sau programatic, introducând o declarație în codul programului. Exemplu:

```
public Form1()
        {
            InitializeComponent();
            this.Text = "Primul formular";
        }
```

Proprietățile ForeColor și BackColor. Prima proprietate enunțată setează culoare textului din formular, iar cea de a doua setează culoarea formularului. Toate acestea le puteți modifica după preferințe din fereastra Properties.



Proprietatea BorderStyle. Controlează stilul bordurii unui formular. Încercați să vedeți cum se modifică setând proprietatea la Fixed3D. (tot din fereastra Properties)
Proprietatea FormatString vă permite să setați un format comun de afișare pentru toate obiectele din cadrul unei ListBox. Aceasta se găsește disponibilă în panoul Properties.
Proprietatea Multiline schimbă setarea implicită a controlului TextBox de la o singură linie, la mai multe linii. Pentru a realiza acest lucru trageți un TextBox într-un formular și modificați valoarea proprietății Multiline din panoul Properties de la False la true.
Proprietatea AutoCheck când are valoarea true, un buton radio își va schimba starea automat la executarea unui clic.
Proprietatea AutoSize folosită la controalele Label și Picture, decide dacă un control este redimensionat automat, pentru a-i cuprinde întreg conținutul.
Proprietatea Enabled determină dacă un control este sau nu activat într-un formular.
Proprietatea Font determină fontul folosit într-un formular sau control.
Proprietatea ImageAlign specifică alinierea unei imagini așezate pe suprafața controlului.
Proprietatea TabIndex setează sau returnează poziția controlului în cadrul aranjării taburilor.
Proprietatea Visible setează vizibilitatea controlului.
Proprietatea Width and Height permite setarea înălțimii și a lățimii controlului.

5.4.3. Câteva dintre metodele și evenimentele Form

Când dezvoltăm programe pentru Windows, uneori trebuie să afișăm ferestre adiționale. De asemenea trebuie să le facem să dispară de pe ecran. Pentru a reuși acest lucru folosim metodele `Show()` și `Close()` ale controlului. Cel mai important eveniment pentru **Button** este **Click** (desemnând acțiunea clic stânga pe buton).

Un exemplu în acest sens:

Deschideți o nouă aplicație Windows. Trageți un buton pe formular. Din meniul Project selectați Add Windows Form, iar în caseta de dialog care apare adăugați numele Form2, pentru noul formular creat. În acest moment ați inclus în program două formulare. Trageți un buton în Form2 și executați dublu clic pe buton, pentru a afișa administratorul său de evenimente. Introduceți acum în el linia de cod

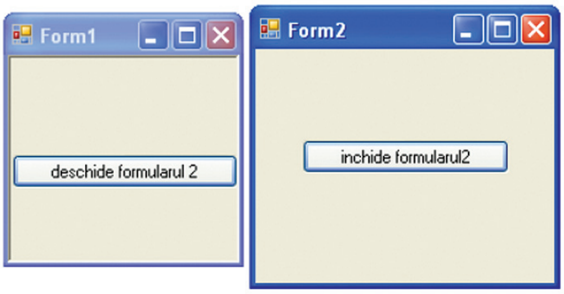
```
this.Close();.
        private void button1_Click(object sender, EventArgs e)
        {
            this.Close();
        }
```

Numele metodei `button1_Click` este alcătuit din numele controlului `button1`, urmat de numele evenimentului: `Click`.

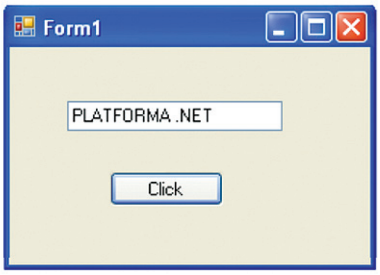
Acum ar trebui să reveniți la Form1 și executați dublu clic pe butonul din acest formular pentru a ajunge la administratorul său de evenimente. Editați administratorul evenimentului conform exemplului de mai jos:

```
private void button1_Click(object sender, EventArgs e)
        {
            Form2 form2 = new Form2();
            form2.Show();
        }
```

În acest moment rulați programul apăsând tasta F5 și veți observa că la executarea unui clic pe butonul din Form1 se deschide Form2 iar la executarea unui clic pe butonul din Form2 acesta se închide.



Tot în cadrul evenimentului Click, oferim acum un exemplu de afișare într-un TextBox a unui mesaj, în momentul în care se execută clic pe un buton:



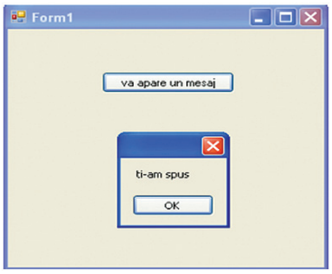
Deschideți o nouă aplicație Windows. Trageți un buton pe formular și o casetă TextBox. Modificați textul ce apare pe buton, conform imaginii, și executați dublu clic pe el, pentru a ajunge la administratorul său de evenimente. Modificați sursa astfel încât să arate în modul următor.

```
private void button1_Click(object sender, EventArgs e)
{
    string a = "PLATFORMA .NET";
        textBox1.Text = a;
}
```

Casete de mesaje:

Pentru a crea o casetă mesaj, apelăm metoda `MessageBox.Show();`.Într-o nouă aplicație Windows, trageți un buton în formular, modificați textul butonului cum doriți sau ca în imaginea alăturată „va apare un mesaj", executați dublu clic pe buton și

adăugați în administratorul evenimentului Click linia de program: `MessageBox.Show("ti-am spus");`. Apoi rulați programul.



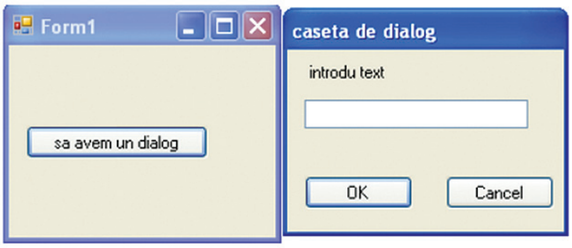
Casete de dialog

O casetă de dialog este o formă specializată de control de tip **Form**.

Exemplu:

Creați o nouă aplicație Windows, apoi trageți un buton în formular și setați proprietatea Text a butonului la : „să avem un dialog", iar apoi executați dublu clic pe buton și folosiți următorul cod pentru administratorul evenimentului **Click.**

```
private void button1_Click(object sender, EventArgs e)
        {
            Form2 w = new Form2();
            w.ShowDialog();
        }
```



Creați un alt formular la acest proiect(alegeți Add Windows Forms din meniul Project), apoi în ordine: setați proprietatea ControlBox la valoarea false, setați proprietatea Text la "casetă de dialog", trageți în formular un control de tip Label și Text la "introdu text", adăugați un control TextBox în formular, adăugați două butoane, setați proprietatea Text a butonului din stânga la "OK" iar al celui din dreapta la "Cancel", setați proprietatea DialogResult a butonului din stanga la OK iar al celui din dreapta la Cancel, executați clic pe formularul casetei de dialog și setați proprietatea AcceptButton la OKButton iar proprietatea CancelButton la CancelButton. Acum executați dublu clic pe butonul OK și folosiți următorul cod pentru administratorul evenimentului Click:

```
private void button1_Click(object sender, EventArgs e)
        {
            textBoxText = textBox1.Text;
            this.Close();
        }
```

Executați dublu clic pe butonul Cancel și folosiți următorul cod pentru administratorul evenimentului Click:

```
private void button2_Click(object sender, EventArgs e)
        {
            Form2 v = new Form2();
            v.ShowDialog();
            if (v.DialogResult != DialogResult.OK)
            { this.textBox1.Clear(); }
        }
```

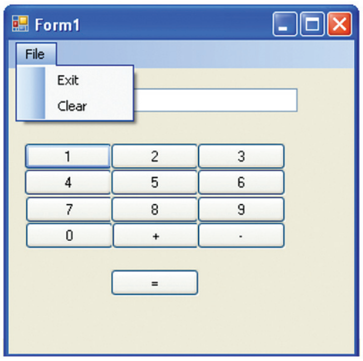
La începutul clasei `Form2` adăugați declarația: `public string textBoxText;`
Iar la sfârșitul clasei `Form2` adăugați proprietatea:

```
public string TextBoxText
        {get{ return(textBoxText);}
```

Acum puteți rula acest program.

Crearea interfeței cu utilizatorul:

Vom crea o aplicației numită Minicalculator, ce va conține un meniu principal. Meniul principal va avea un obiect **File** ce va conține câte un obiect **Exit** și **Clear**. Ieșirile vor fi afișate într-un **TextBox**.



Creați o nouă aplicație Windows în care trageți 13 butoane pe care le veți poziționa și numi ca în figura alăturată, apoi mai adăugați un TextBox(pe acesta îl puteți seta să arate textul în stânga sau în dreapta).

Adăugați un menuStrip care să conțină elementele precizate mai sus, și pe care le puteți observa în figura alăturată.

Faceți dublu clic pe fiecare buton numeric în parte pentru a ajunge la sursă și modificați fiecare sursă respectând codul de mai jos:

```
private void button7_Click(object sender, EventArgs e)
        {
            string v = textBox1.Text;
            v += "7";
            textBox1.Text = v;
        }
```

Am dat exemplu pentru tasta 7, dar atenție la fiecare tastă, variabila v, va primi ca valoare numărul afișat pe tasta respectivă.

Acum procedați la fel, și modificați sursa pentru butoanele + și -.

```
  private void button11_Click(object sender, EventArgs e)
        {
            op1 = textBox1.Text;
            operatie = "+";
            textBox1.Text = "";
        }
```

Pentru butonul = folosiți codul următor:

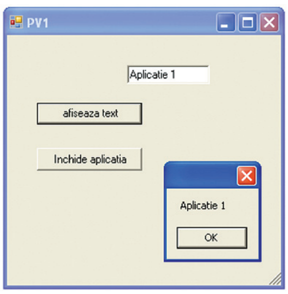
```
private void button13_Click(object sender, EventArgs e)
        {
            int n1, n2, x = 0;
            n1 = Convert.ToInt16(op1);
            n2 = Convert.ToInt16(op2);
            if (operatie == "+") { x = n1 + n2; }
            else if (operatie == "-") { x = n1 - n2; }
            textBox1.Text = Convert.ToString(x);
            op1 = ""; op2 = "";
        }
```

Codul pentru obiectele Exit și Clear din meniul File va fi:

```
private void clearToolStripMenuItem_Click(object sender, EventArgs e)
        {
            textBox1.Text = "";
        }
        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this.Close();
        }
```

Un alt exemplu:

Se adaugă pe formular două butoane și o casetă text. Apăsarea primului buton va determina afișarea textului din TextBox într-un MessageBox iar apăsarea celui de-al doilea buton va închide închide aplicația.



După adăugarea celor două butoane și a casetei text, a fost schimbat textul afișat pe cele două butoane și au fost scrise funcțiile de tratare a evenimentului **Click** pentru cele două butoane:

```
private void button1_Click(object sender, System.EventArgs e)
          { MessageBox.Show(textBox1.Text);}
private void button2_Click(object sender, System.EventArgs e)
          { Form1.ActiveForm.Dispose();}
```

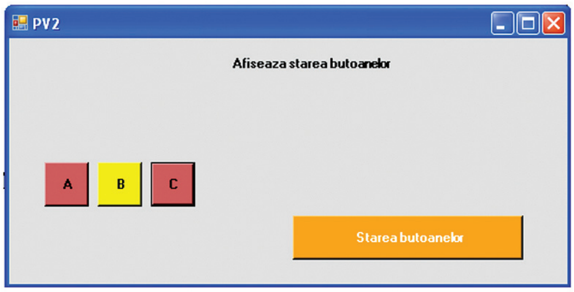
- **Controale valoare** (**label, textbox, picturebox**) care arată utilizatorului o informație (text, imagine).

  Label este folosit pentru plasarea de text pe un formular. Textul afișat este conținut în propietatea Text și este aliniat conform propietății TextAlign.

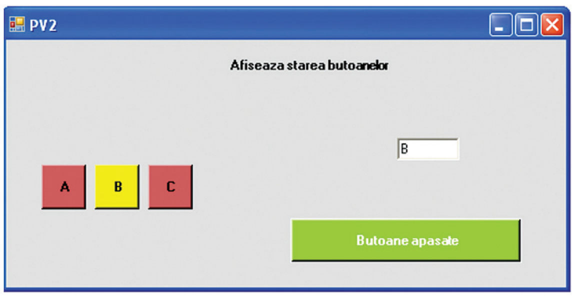
  TextBox - permite utilizatorului să introducă un text. Prevede, prin intermediul ContextMenu-ului asociat, un set de funcționalități de bază, ca de exemplu (Cut, Copy, Paste, Delete, SelectAll).

  PictureBox permite afișarea unei imagini.

Exemplul PV2 afișează un grup alcătuit din 3 butoane, etichetate A,B respectiv C având inițial culoarea roșie. Apăsarea unui buton determină schimbarea culorii acestuia în galben. La o nouă apăsare butonul revine la culoare inițială. Acționarea butonului "Starea butoanelor" determină afișarea într-o casetă text a etichetelor butoanelor galbene. Caseta text devine vizibilă atunci când apăsăm prima oară acest buton. Culoarea butonului mare (verde/portocaliu) se schimbă atunci când mouse-ul este poziționat pe buton.



După adăugarea butoanelor și a casetei text pe formular, stabilim evenimentele care determină schimbarea culoriilor și completarea casetei text.



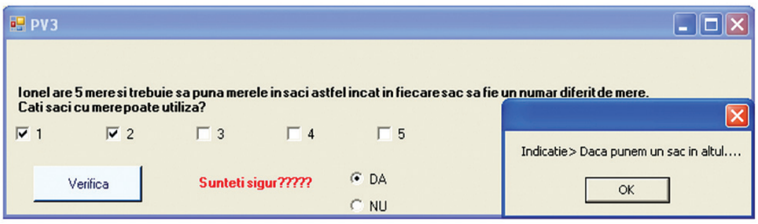
```
private void button1_Click(object sender, System.EventArgs e)
{if (button1.BackColor== Color.IndianRed) button1.BackColor=Color.Yellow;
 else button1.BackColor= Color.IndianRed;}
private void button4_MouseEnter(object sender, System.EventArgs e)
{button4.BackColor=Color.YellowGreen;button4.Text="Butoane apasate";}
private void button4_MouseLeave(object sender, System.EventArgs e)
{textBox1.Visible=false;button4.Text="Starea butoanelor";
 button4.BackColor=Color.Orange;}
private void button4_Click(object sender, System.EventArgs e)
{textBox1.Visible=true;textBox1.Text="";
    if( button1.BackColor==Color.Yellow)textBox1.Text=textBox1.Text+'A';
    if( button2.BackColor==Color.Yellow)textBox1.Text=textBox1.Text+'B';
    if( button3.BackColor==Color.Yellow)textBox1.Text=textBox1.Text+'C';
}
```

<u>Exercițiu</u> Modificați aplicația precedentă astfel încât să avem un singur eveniment button_Click, diferențierea fiind făcută de parametrul sender.

<u>Exercițiu (Password)</u> Adăugați pe un formular o casetă text în care să introduceți un șir de caractere și apoi verificați dacă acesta coincide cu o parolă dată. Textul introdus în casetă nu este vizibil (fiecare caracter este înlocuit cu*). Rezultatul va fi afișat într-un MessageBox.

- **Controale de selecție (CheckBox,RadioButton)** au propietatea **Checked** care indică dacă am selectat controlul. După schimbarea stării unui astfel de control, se declanșează evenimentul **Checked.** Dacă propietatea **ThreeState** este setată, atunci se schimbă funcționalitatea acestor controale, în sensul că acestea vor permite setarea unei alte stări. În acest caz, trebuie verificată propietatea **CheckState(Checked**, **Unchecked,Indeterminate**) pentru a vedea starea controlului.

Aplicația PV3 este un exemplu de utilizare a acestor controale. Soluția unei probleme cu mai multe variante de răspuns este memorată cu ajutorul unor checkbox-uri cu proprietatea ThreeState. Apăsarea butonului *Verifică* determină afișarea unei etichete și a butoanelor radio DA și NU. Răspunsul este afișat într-un MessageBox.



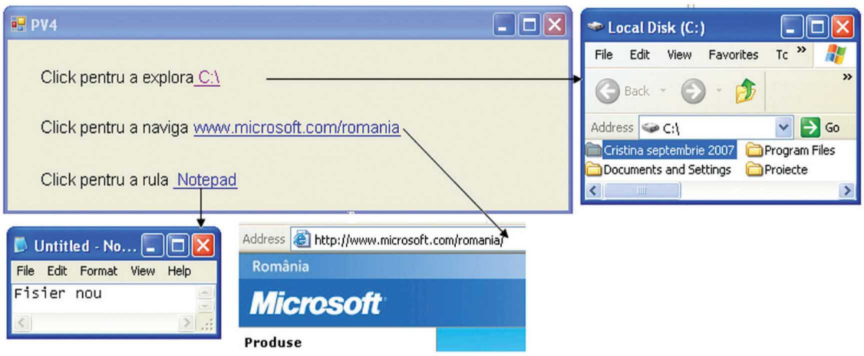
După adăugarea controalelor pe formular și setarea proprietăților **Text** și **ThreeState** în cazul checkbox-urilor stabilim evenimentele click pentru butonul Verifica și pentru butonul radio cu eticheta DA:

```
private void radioButton1_Click(object sender, System.EventArgs e)
{if (checkBox1.CheckState==CheckState.Checked &&
    checkBox2.CheckState==CheckState.Checked &&
    checkBox3.CheckState==CheckState.Checked &&
    checkBox5.CheckState==CheckState.Checked &&
    checkBox4.CheckState==CheckState.Unchecked) MessageBox.Show("CORECT");
 else MessageBox.Show("Indicatie> Daca punem un sac in altul....");
 label2.Visible=false;
 radioButton1.Checked=false; radioButton2.Checked=false;
 radioButton1.Visible=false; radioButton2.Visible=false;}
private void button1_Click(object sender, System.EventArgs e)
{label2.Visible=true;radioButton1.Visible=true;radioButton2.Visible=true;}
```

Exercițiu (Test grilă) Construiți un test grilă care conține 5 itemi cu câte 4 variante de răspuns (alegere simplă sau multiplă), memorați răspunsurile date și afișați, după efectuarea testului, într-o casetă text, în dreptul fiecărui item, răspunsul corect.

- **LinkLabel** afișează un text cu posibilitatea ca anumite părți ale textului (**LinkArea)** să fie desenate ca și hyperlink-uri. Pentru a face link-ul funcțional trebuie tratat evenimentul **LinkClicked.**

În exemplul **PV4**, prima etichetă permite afișarea conținutului discului C:, a doua legătură este un link către pagina www.microsoft.com/romania și a treia accesează Notepad.

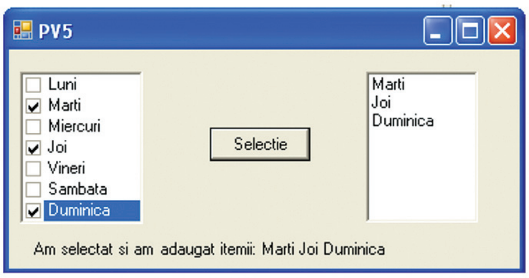



```
private void etichetaC_LinkClicked (object sender,
                                    LinkLabelLinkClickedEventArgs e )
{ etichetaC.LinkVisited = true;
  System.Diagnostics.Process.Start( @"C:\" );}
private void etichetaI_LinkClicked( object sender,
                                    LinkLabelLinkClickedEventArgs e )
{etichetaI.LinkVisited = true;
 System.Diagnostics.Process.Start("IExplore",
                               "http://www.microsoft.com/romania/" );}
private void etichetaN_LinkClicked( object sender,
                                     LinkLabelLinkClickedEventArgs e )
{etichetaN.LinkVisited = true;
 System.Diagnostics.Process.Start( "notepad" );}
```

Exercițiu (Memorator) Construiți o aplicație care să conțină patru legături către cele patru fișiere/ pagini care conțin rezumatul capitolelor studiate.

Controale pentru listare (**ListBox**, **CheckedListBox**, **ComboBox, ImageList**) ce pot fi legate de un DataSet, de un ArrayList sau de orice tablou (orice sursă de date ce implementează interfața IEnumerable).

În exemplul **PV5** elementele selectate din **CheckedListBox** se adaugă în **ListBox.** După adăugarea pe formular a CheckedListBox-ului, stabilim colecția de itemi (Properties-Items-Collection), butonul Selecție și ListBox-ul.



Evenimentul **Click** asociat butonului **Setectie** golește mai întâi listBox-ul (`listBox1.Items.Clear();`) și după aceea adaugă în ordine fiecare element selectat din CheckedListBox. Suplimentar se afișează o etichetă cu itemii selectați.

```
void button1_Click(object source, System.EventArgs e)
{  String s = "Am selectat si am adaugat itemii: ";
   listBox1.Items.Clear();
   foreach ( object c in  checkedListBox1.CheckedItems)
    {listBox1.Items.Add(c);
      s = s + c.ToString();s = s + " ";}
   label1.Text = s;}
```

Exercițiu (Filtru) Construiți o aplicație care afișează fișierele dintr-un folder ales care au un anumit tip (tipul fișierelor este ales de utilizator pe baza unui CheckedListBox)



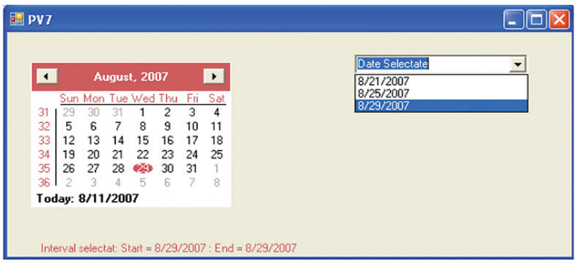
Aplicația **PV6** este un exemplu de utilizare a controlului **ImageList.** Apăsarea butonului **Desene** va adăuga fișierele *.gif din folderul **C:\Imagini** în listă și va afișa conținutul acesteia. Butonul **Animate** va determina afișarea fișierelor *.gif cu ajutorul PictureBox.

```
ImageList desene_animate = new System.Windows.Forms.ImageList();
private void contruieste_lista_Click(object sender, System.EventArgs e)
{    // Configureaza lista
     desene_animate.ColorDepth =System.Windows.Forms.ColorDepth.Depth8Bit;
     desene_animate.ImageSize = new System.Drawing.Size(60, 60);
     desene_animate.Images.Clear();
     string[] gif_uri = Directory.GetFiles("C:\\Imagini", "*.gif");
// se construieste un obiect Imagine pentru fiecare fisier si se adauga la
ImageList.
     foreach (string fisier_gif in gif_uri)
     {Bitmap desen= new Bitmap (fisier_gif);
      desene_animate.Images.Add(desen);pictureBox2.Image=desen;}

      Graphics g = this.CreateGraphics();
   // Deseneaza fiecare imagine utilizand metoda  ImageList.Draw()
      for (int i = 0; i < desene_animate.Images.Count; i++)
       desene_animate.Draw(g, 60 + i * 60, 60, i);
     g.Dispose();
}
```

<u>Exercițiu (Thumbnails)</u> Afișați într-o ferestră conținutul folder-ului curent în mod View-Thumbnails.

MonthCalendar afișează un calendar prin care se poate selecta o dată (zi, luna, an) în mod grafic. Proprietățile mai importante sunt: **MinDate**, **MaxDate**, **TodayDate** ce reprezintă data minimă/maximă selectabilă și data curentă (care apare afișată diferențiat sau nu în funcție de valorile proprietăților **ShowToday**,**ShowTodayCircle**. Există 2 evenimente pe care controlul le expune: **DateSelected** și **DateChanged**. În rutinele de tratare a acestor evenimente, programatorul are acces la un obiect de tipul **DateRangeEventArgs** care conține proprietățile **Start** și **End** (reprezentând intervalul de timp selectat).



Formularul din aplicația **PV7** conține un calendar pentru care putem selecta un interval de maximum 30 de zile, sunt afișate săptămânile și ziua curentă. Intervalul selectat se afișează prin intermediul unei etichete. Dacă se selectează o dată atunci aceasta va fi adăugată ca item într-un ComboBox (orice dată poate apărea cel mult o dată în listă).

După adăugarea celor 3 controale pe formular, stabilim proprietățile pentru monthCalendar1 (**ShowWeekNumber**-True, **MaxSelectionCount**-30, etc.) și precizăm ce se execută atunci când selectăm un interval de timp:

```
private void monthCalendar1_DateSelected(object sender,
                              System.Windows.Forms.DateRangeEventArgs e)
{ this.label1.Text = "Interval selectat: Start = "
                     +e.Start.ToShortDateString() + " : End = "
                     + e.End.ToShortDateString();
if (e.Start.ToShortDateString()==e.End.ToShortDateString())
    {String x=e.Start.ToShortDateString();
     if(!(comboBox1.Items.Contains(x)))
     comboBox1.Items.Add(e.End.ToShortDateString());}
}
```

- **DateTimePicker** este un control care (ca și MonthCalendar) se poate utiliza pentru a selecta o dată. La clic se afișează un control de tip MonthCalendar, prin care se poate selecta data dorită. Fiind foarte asemănător cu MonthCalendar, proprietățile prin care se poate modifica comportamentul controlului sunt identice cu cele ale controlului MonthControl.

Exercițiu (Formular) Contruiți un formular de introducere a datelor necesare realizării unei adrese de e-mail. Data nașterii va fi selectată direct utilizând MonthCalendar.

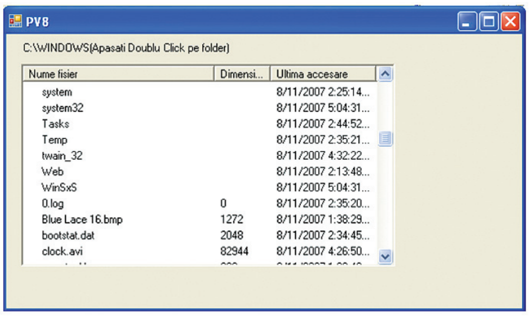
- **ListView** este folosit pentru a afișa o colecție de elemente în unul din cele 4 moduri (**Text, Text+Imagini mici, Imagini mari, Detalii**). Acesta este similar grafic cu ferestrele în care se afișează fișierele dintr-un anumit director din Windows Explorer. Fiind un control complex, conține foarte multe proprietăți, printre care: **View** (selectează modul de afișare (LargeIcon, SmallIcon, Details, List)), **LargeImageList, SmallImageList** (icon-urile de afișat în modurile LargeIcon, SmallIcon), **Columns**(utilizat doar în modul Details, pentru a defini coloanele de afișat), **Items**(elementele de afișat).

Aplicația **PV8** este un exemplu de utilizare *ListView*. Se pornește de la rădăcină și se afișează conținutul folder-ului selectat cu dublu clic. La expandare se afișează numele complet, data ultimei accesări și, în cazul fișierelor, dimensiunea.

Controlul **lista_fisiere** este de tip **ListView**.

Funcția `ConstruiesteHeader` permite stabilirea celor trei coloane de afișat.

```
private void ConstruiesteHeader()
         {ColumnHeader colHead;colHead = new ColumnHeader();
          colHead.Text = "Nume fisier";
this.lista_fisiere.Columns.Add(colHead);
           colHead = new ColumnHeader();colHead.Text = "Dimensiune";
           his.lista_fisiere.Columns.Add(colHead);
           colHead = new ColumnHeader();colHead.Text = "Ultima accesare";
           this.lista_fisiere.Columns.Add(colHead);
           }
```



Pentru item-ul selectat se afișează mai întâi folderele și după aceea fișierele. Pentru aceasta trebuie să determinăm conținutul acestuia:

```
ListViewItem lvi;
ListViewItem.ListViewSubItem lvsi;
    this.calea_curenta.Text = radacina + "(Doublu Click pe folder)";
System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(radacina);
DirectoryInfo[] dirs = dir.GetDirectories();
FileInfo[] files = dir.GetFiles();
```

să ștergem vechiul conținut al listei:

```
this.lista_fisiere.Items.Clear();
this.lista_fisiere.BeginUpdate();
```

și să adăugăm fiecare nou item (coloana a doua este vidă în cazul folderelor):

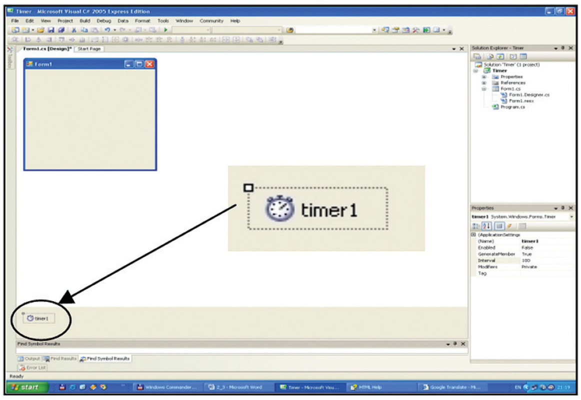
```
foreach (System.IO.DirectoryInfo fi in dirs)
        { lvi = new ListViewItem();lvi.Text = fi.Name;
       lvi.ImageIndex = 1; lvi.Tag = fi.FullName;
          lvsi = new ListViewItem.ListViewSubItem();
          lvsi.Text = "";lvi.SubItems.Add(lvsi);
          lvsi = new ListViewItem.ListViewSubItem();
         lvsi.Text = fi.LastAccessTime.ToString();
         lvi.SubItems.Add(lvsi); this.lista_fisiere.Items.Add(lvi);
        }
```

<u>Exercițiu (Ordonare)</u> Modificați aplicația anterioară astfel încât apăsarea pe numele unei coloane să determine afișarea informațiilor ordonate după criteriul specificat (nume, dimensiune, data).

- Controale "de control" al executării (**Timer**) sau de dialog (**OpenFileDialog**, **SaveFileDialog**, **ColorDialog**, **FontDialog**, **ContextMenu**).

Utilizatorul nu are drept de control asupra tuturor controalelor. Dintre acestea vom studia în cele ce urmează controlul **Timer** asupra căruia are drept de interacțiune doar cel care dezvoltă aplicația.

Observăm că aducând din **Toolbox** controlul **Timer**, acesta nu se afișează pe formular, el apărând într-o zonă gri a suprafeței de lucru (**Designer**).



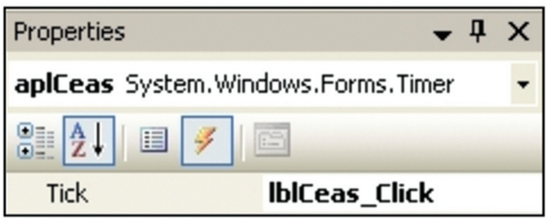
Vom stabili următoarele proprietăti legate de **Timer**:

Proprietate	Valoare	Explicație
(Name)	aplCeas	
Enabled	True	Activarea controlului de timp
Interval	1.000	Numărul de milisecunde dintre apelurile la metoda de tratare a evenimentului. Se stabilește, în cazul de față numărătoarea din secundă în secundă

Aducem în formular un control **Label** cu următoarele proprietăți:

Control	Proprietate	Valoare
label1	(Name)	labelCeas
	AutoSize	False
	BorderStyle	Fixed3D
	FontSize	16,25, Bold
	Location	82;112
	Text	
	Size	129;42
	TextAlign	MiddleCenter

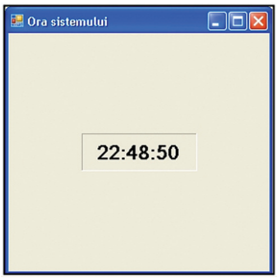
Dăm clic pe icoana de la **timer** care are numele **aplCeas**, iar la **Events**, la **Tick** selectăm **lblCeas_Click**



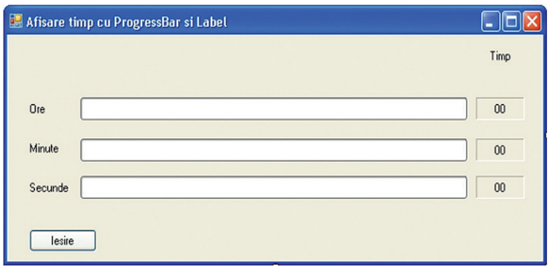
Dăm dublu clic pe **label** și inserăm codul:

```
private void lblCeas_Click(object sender, EventArgs e)
        {
            DateTime OraCurenta = DateTime.Now;
            lblCeas.Text=OraCurenta.ToLongTimeString();
        }
```

Compilăm și obținem într-o fereastră vizualizarea orei sistemului.



În următorul exemplu vom folosi **ProgressBar** pentru a vizualiza ceasul sistemului. Vom construi un formular ca în imaginea alăturată. Pentru aceasta aducem din **Toolbox** trei controale **ProgressBar**, un control **Timer**, șapte controale **Label** și un control **Button**. Tabelul de mai jos descrie proprietățile și valorile formularului și a respectivelor controale:



Formularul:

Control	Proprietate	Valoare
Form1	(Name)	Form1
	Size	606;265
	BorderStyle	Fixed3D
	Text	Afișare timp cu ProgressBar si Label

ProgressBar-urile:

Control	Proprietate	Valoare
ProgressBar1	(Name)	prgOre
	Location	82;64
	Maximum	23
	Size	431;23
	Step	1
	Style	Blocks

Control	Proprietate	Valoare
ProgressBar2	(Name)	prgMinute
	Location	82;106
	Maximum	59
	Size	431;23
	Step	1
	Style	Blocks
ProgressBar3	(Name)	prgSecunde
	Location	82;144
	Maximum	59
	Size	431;23
	Step	1
	Style	Blocks

Pentru afișarea textelor „Ore”, „Minute”, „Secunde” folosim:

Control	Proprietate	Valoare
Label1	(name)	lblOre
	Autosize	False
	BorderStyle	None
	Locations	22;64
	Size	54;23
	Text	Ore
	TextAlign	MiddleLeft
Label2	(name)	lblMinute
	Autosize	False
	BorderStyle	None
	Locations	22;104
	Size	54;23
	Text	Minute
	TextAlign	MiddleLeft
Label3	(name)	lblSecunde
	Autosize	False
	BorderStyle	None
	Locations	22;144
	Size	54;23
	Text	Minute
	TextAlign	MiddleLeft

Pentru „Ieșire” folosim:

Control	Proprietate	Valoare
Button1	(Name)	btnIesire
	Location	25;198
	Size	75;23
	Text	Iesire

Pentru informațiile din partea dreaptă a formularului:

Control	Proprietate	Valoare
Label4	(name)	lblTextTimp
	Autosize	False
	Locations	523;10
	Size	54;23
	Text	Timp
	TextAlign	MiddleCenter
Label5	(name)	lblAfisOre
	Autosize	False
	BorderStyle	Fixed3D
	Locations	523;64
	Size	54;23
	Text	00
	TextAlign	MiddleCenter
Label6	(name)	lblAfisMinute
	Autosize	False
	BorderStyle	Fixed3D
	Locations	523;106
	Size	54;23
	Text	00
	TextAlign	MiddleCenter
Label4	(name)	lblAfisSecunde
	Autosize	False
	BorderStyle	Fixed3D
	Locations	523;144
	Size	54;23
	Text	00
	TextAlign	MiddleCenter

Pentru **Timer**:

Control	Proprietate	Valoare	Evenimente
Timer1	(Name)	Timer1	Timer1_Tick
	Enabled	True	
	Interval	1.000	

pentru **timer1**:

```
private void timer1_Tick(object sender, EventArgs e)
        {
            DateTime TimpCurent = DateTime.Now;
            int H = TimpCurent.Hour;
            int M = TimpCurent.Minute;
            int S = TimpCurent.Second;
            prgOre.Value = H;
            prgMinute.Value = M;
            prgSecunde.Value = S;

            lblAfisOre.Text = H.ToString();
```

```
            lblAfisMinute.Text = M.ToString();
            lblAfisSecunde.Text = S.ToString();
        }
```

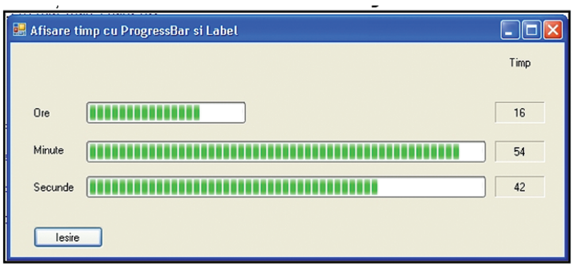
pentru a redimensiona proporțional **ProgressBar**-ul Ore cu cel care reprezintă Minutele, respectiv Secundele, introducem următorul cod:

```
private void Form1_Load(object sender, EventArgs e)
        {
            this.prgOre.Width = 2 * this.prgMinute.Width
                                                        / 5;
        }
```

pentru butonul de Ieșire:

```
private void btnIesire_Click(object sender, EventArgs e)
        {
            Close();
        }
```

Compilăm și obținem ora sistemului afișată într-o fereastră de forma:

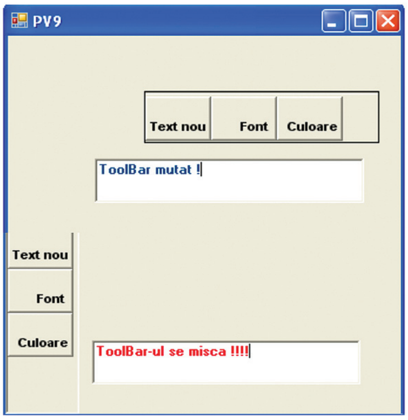


- Grupuri de controale **Toolbar** (*ToolStrip*) afișează o bară de butoane în partea de sus a unui formular. Se pot introduce vizual butoane (printr-un designer, direct din Visual Studio.NET IDE), la care se pot seta atât textul afișat sau imaginea. Evenimentul cel mai util al acestui control este **ButtonClick** (care are ca parametru un obiect de tip **ToolBarButtonClickEventArgs**, prin care programatorul are acces la butonul care a fost apasat).

În aplicația următoare **PV9** cele 3 butoane ale toolbar-ului permit modificarea proprietăților textului introdus în casetă. Toolbar-ul se poate muta fără a depăși spațiul ferestrei. Schimbarea fontului se realizează cu ajutorul unui control FontDialog(),iar schimbarea culorii utilizează ColorDialog()

```
FontDialog fd = new FontDialog();
fd.ShowColor = true;fd.Color = Color.IndianRed;
fd.ShowApply = true;
fd.Apply += new EventHandler(ApplyFont);
if(fd.ShowDialog() !=
      System.Windows.Forms.DialogResult.Cancel)
```

```
 { this.richTextBox1.Font= fd.Font;
   this.richTextBox1.ForeColor=fd.Color;
 }
ColorDialog cd = new ColorDialog();
cd.AllowFullOpen = true;cd.Color = Color.DarkBlue;
if(cd.ShowDialog() == System.Windows.Forms.DialogResult.OK)
this.richTextBox1.ForeColor = cd.Color;
```



Mutarea toolbar-ul este dirijată de evenimentele produse atunci când apăsăm butonul de mouse și/sau ne deplasăm pe suprafața ferestrei.

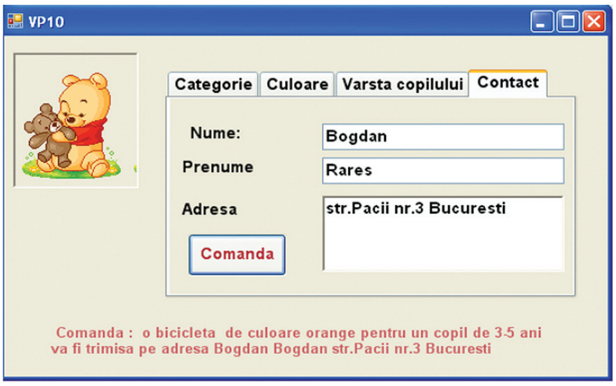
```
private void toolBar1_MouseDown(object sender, MouseEventArgs e)
{ // am apasat butonul de mouse pe toolbar
   am_apasat = true;
   forma_deplasata = new Point(e.X, e.Y); toolBar1.Capture = true;}
private void toolBar1_MouseUp(object sender, MouseEventArgs e)
{  am_apasat = false;toolBar1.Capture = false;}
private void toolBar1_MouseMove(object sender, MouseEventArgs e)
{  if (am_apasat)
   { if(toolBar1.Dock == DockStyle.Top || toolBar1.Dock == DockStyle.Left)
     { // daca depaseste atunci duc in stanga sus
       if (forma_deplasata.X < (e.X-20) || forma_deplasata.Y < (e.Y-20))
  {  am_apasat = false;// Disconect toolbar
     toolBar1.Dock = DockStyle.None;
     toolBar1.Location = new Point(10, 10);
     toolBar1.Size = new Size(200, 45);
     toolBar1.BorderStyle = BorderStyle.FixedSingle;
  }
}
     else if (toolBar1.Dock == DockStyle.None)
  {toolBar1.Left = e.X + toolBar1.Left - forma_deplasata.X;
   toolBar1.Top = e.Y + toolBar1.Top - forma_deplasata.Y;
   if (toolBar1.Top < 5 || toolBar1.Top>this.Size.Height-20)
   {   am_apasat = false;toolBar1.Dock = DockStyle.Top;
       toolBar1.BorderStyle = BorderStyle.Fixed3D;}
   else if (toolBar1.Left < 5 || toolBar1.Left > this.Size.Width - 20)
        {  am_apasat = false;toolBar1.Dock = DockStyle.Left;
```

```
            toolBar1.BorderStyle = BorderStyle.Fixed3D;
        }}}
}
```

Exercițiu (Editor) Realizați un editor de texte care conțină un control toolBar cu butoanele uzuale.

- **Controale container** (**GroupBox, Panel, TabControl**) sunt controale ce pot conține alte controale.

Aplicația **PV10** simulează lansarea unei comenzi către un magazin de jucării. Se utilizează 4 pagini de Tab pentru a simula selectarea unor opțiuni ce se pot grupa pe categorii.



Exercițiu (Magazin) Dezvoltați aplicația precedentă astfel încât pe o pagină să se afișeze modelele disponibile (imagine+detalii) și să se permită selectarea mai multor obiecte. Ultima pagină reprezintă coșul de cumpărături.

- **Grupuri de controale tip Meniu** (*MenuStrip, ContextMenuStrip etc.*)

Un formular poate afișa un singur meniu principal la un moment dat, meniul asociat inițial fiind specificat prin propietatea **Form.MainMenuStrip**. Meniul care este afișat de către un formular poate fi schimbat dinamic la rulare:

```
switch(cond) { case cond1:this.MainMenuStrip = this.mainMenu1;break;
                case cond2:this.MainMenuStrip = this.mainMenu2;
              }
```

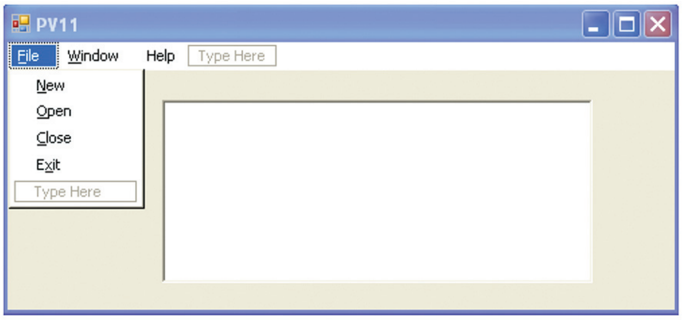
unde mainMenu1 și mainMenu2 sunt obiecte de tip **MenuStrip**. Editarea unui astfel de obiect se poate face utilizând **Menu Designer**. Clasa **MenuStrip** are o colecție de **MenuItem** care conține 0 sau mai multe obiecte de tip **MenuItem**. Fiecare dintre aceste obiecte de tip MenuItem are 0 sau mai multe obiecte de tip MenuItem, care vor constitui noul nivel de itemi (Ex: File →New,Save, Open, Close, Exit).

Propietățile Checked și **RadioCheck** indică itemul selectat, **Enabled** and **Visible** determină dacă un item poate fi sau nu selectat sau vizibil, **Shortcut** permite asignarea unei combinații de taste pentru selectarea unui item al meniului și **Text**

memorează textul care va fi afișat pentru respectivul item al meniului.

Evenimentul **Click** are loc când un utilizator apasă un item al meniului.

Exemplul **PV11** permite, prin intermediul unui meniu, scrierea unui fișier Notpad, afișarea continutului acestuia într-o casetă text, schimbarea fontului și culorii de afișare, ștergerea conținutului casetei, afișarea unor informații teoretice precum și Help dinamic. Au fost definite chei de acces rapid pentru accesarea componentelor meniului.



File→ New permite scrierea unui fișier notepad nou

```
System.Diagnostics.Process.Start( "notepad" );
```

File→ Open selectează și afișează în caseta text conținutul unui fișier text.

```
OpenFileDialog of = new OpenFileDialog();
of.Filter = "Text Files (*.txt)|*.txt";
of.Title = "Fisiere Text";
if (of.ShowDialog() == DialogResult.Cancel)return;
richTextBox1.Text="";richTextBox1.Visible=true;
FileStream strm;
try{strm = new FileStream (of.FileName, FileMode.Open, FileAccess.Read);
    StreamReader rdr = new StreamReader (strm);
    while (rdr.Peek() >= 0){string str = rdr.ReadLine ();
                          richTextBox1.Text=richTextBox1.Text+" "+str;}
   }
catch (Exception){MessageBox.Show ("Error opening file", "File Error",
                  MessageBoxButtons.OK, MessageBoxIcon.Exclamation);}
```

File→ Close șterge conținutul casetei text, File→ Exit închide aplicația
Window → Font și Window →Color permit stabilirea fontului/culorii textului afișat.
Help→ DinamicHelp accesează

```
System.Diagnostics.Process.Start("IExplore",
                        "http://msdn2.microsoft.com/en-us/default.aspx");
```

Help→ About PV afișează în caseta text informații despre implementarea unui menu.

Exerciţiu (Fisiere) Contruiţi un menu care să permită efectuarea operaţiilor uzuale cu fişiere.

5.5. System.Drawing

Spaţiul `System.Drawing` conţine tipuri care permit realizarea unor desene 2D şi au rol deosebit în proiectarea interfeţelor grafice.

Un obiect de tip **Point** este reprezentat prin coordonatele unui punct într-un spaţiu bidimensional (exemplu: `Point myPoint = new Point(1,2);`)

Point este utilizat frecvent nu numai pentru desene, ci şi pentru a identifica în program un punct dintr-un anumit spaţiu. De exemplu, pentru a modifica poziţia unui buton în fereastră putem asigna un obiect de tip *Point* proprietăţii *Location* indicând astfel poziţia colţului din stânga-sus al butonului (`button.Location = new Point(100, 30)`). Putem construi un obiect de tip *Point* pentru a redimensiona un alt obiect.

```
Size mySize = new Size(15, 100);
Point myPoint = new Point(mySize);
System.Console.WriteLine("X: " + myPoint.X + ", Y: " + myPoint.Y);
```

Structura **Color** conţine date, tipuri şi metode utile în lucrul cu culori. Fiind un tip valoare (**struct**) şi nu o clasă, aceasta conţine date şi metode, însă nu permite instanţiere, constructori, destructor, moştenire.

```
Color myColor = Color.Brown;  button1.BackColor = myColor;
```

Substructura `FromArgb` a structurii Color returnează o culoare pe baza celor trei componente ale oricărei culori (red, green, blue).

Clasa **Graphics** este o clasă sigilată reprezentând o arie rectangulară care permite reprezentări grafice. De exemplu, o linie frântă se poate realiza astfel:

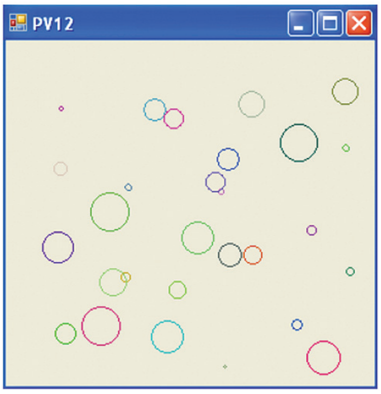
```
Point[] points = new Point[4];
points[0] = new Point(0, 0);points[1] = new Point(0, 120);
points[2] = new Point(20, 120);points[3] = new Point(20, 0);

Graphics g = this.CreateGraphics();
Pen pen = new Pen(Color.Yellow, 2);
g.DrawLines(pen, points);
```

Aplicaţia **PV12** este un exerciţiu care desenează cercuri de raze şi culori aleatorii şi emite sunete cu frecvenţă aleatoare.

```
Random x = new Random();
Console.Beep(300 + x.Next(1000), 150);
Graphics g = e.Graphics;
i = 1 + x.Next(30);
p=new Pen(System.Drawing.Color.FromArgb(x.Next(256),x.Next(256),x.Next(256)))
```

```
g.DrawEllipse(p, x.Next(100), x.Next(100), i, i);
Console.Sleep(200);
```



În exemplul **PV13** se construiește o pictogramă pe baza unei imagini.

```
Image thumbnail;
private void Thumbnails_Load(object sender, EventArgs e)
{ try{Image img = Image.FromFile("C:\\Imagini\\catel.jpg");
      int latime=100, inaltime=100;
      thumbnail=img.GetThumbnailImage(latime, inaltime,null, IntPtr.Zero);}
  catch{MessageBox.Show("Nu exista fisierul");}
}
private void Thumbnails_Paint(object sender, PaintEventArgs e)
  {e.Graphics.DrawImage(thumbnail, 10, 10);}
```



5.6. Validarea informațiilor de la utilizator

Înainte ca informațiile de la utilizator să fie preluate și transmise către alte clase, este necesar să fie validate. Acest aspect este important, pentru a preveni posibilele erori. Astfel, dacă utilizatorul introduce o valoare reală (*float*) când aplicația așteaptă un întreg (*int*), este posibil ca aceasta să se comporte neprevăzut abia câteva secunde mai târziu, și după multe apeluri de metode, fiind foarte greu de identificat cauza primară a problemei.

Validarea la nivel de câmp

Datele pot fi validate pe măsură ce sunt introduse, asociind o prelucrare unuia

dintre handlerele asociate evenimentelor la nivel de control (*Leave*, *Textchanged*, *MouseUp* etc.)

```
private void textBox1_KeyUp(object sender,
                            System.Windows.Forms.KeeyEventArgs e)
{if(e.Alt==true) MessageBox.Show ("Tasta Alt e apasata");     // sau
 if(Char.IsDigit(e.KeyChar)==true)      MessageBox.Show("Ati apasat o cifra");
}
```

Validarea la nivel de utilizator

În unele situații (de exemplu atunci când valorile introduse trebuie să se afle într-o anumită relație între ele), validarea se face la sfârșitul introducerii tuturor datelor la nivelul unui buton final sau la închiderea ferestrei de date.

```
private void btnValidate_Click(object sender, System.EventArgs e)
{  foreach(System.Windows.Forms.Control a in this.Controls)
   {  if( a is System.Windows.Forms.TextBox & a.Text=="")
    { a.Focus();return;}
   }
}
```

ErrorProvider

O manieră simplă de a semnala erori de validare este aceea de a seta un mesaj de eroare pentru fiecare control.

```
myErrorProvider.SetError(txtName," Numele nu are spatii in stanga");
```

<u>Aplicatii recapitulative.</u>

Urmăriți aplicațiile și precizați pentru fiecare dintre ele controalele utilizate, evenimentele tratate: Forma poloneza (PV14), Triunghi (PV15), Ordonare vector(PV16), Subsir crescător de lungime maximă(PV17), Jocul de Nim (PV18)

<u>Exercițiu</u> (Test grila) Realizați un generator de teste grilă (întrebările sunt preluate dintr-un fișier text, pentru fiecare item se precizează tipul (alegere simplă/multiplă), punctajul, enunțul și distractorii, imaginea asociată (dacă există). După efectuarea testului se afișează rezultatul obținut și statistica răspunsurilor.

CAPITOLUL 6

ADO.NET

ADO.NET (**A**ctiveX **D**ata **O**bjects) reprezintă o parte componentă a nucleului .NET Framework ce permite conectarea la surse de date diverse, extragerea, manipularea și actualizarea datelor.

De obicei, sursa de date este o bază de date, dar ar putea de asemenea să fie un fișier text, o foaie Excel, un fișier Access sau un fișier XML.

În aplicațiile tradiționale cu baze de date, clienții stabilesc o conexiune cu baza de date și mențin această conexiune deschisă până la încheierea executării aplicației.

Conexiunile deschise necesită alocarea de resurse sistem. Atunci când menținem mai multe conexiuni deschise server-ul de baze de date va răspunde mai lent la comenzile clienților întrucât cele mai multe baze de date permit un număr foarte mic de conexiuni concurente.

ADO.NET permite și lucrul în stil conectat dar și lucrul în **stil deconectat**, aplicațiile conectându-se la server-ul de baze de date numai pentru extragerea și actualizarea datelor. Acest lucru permite **reducerea numărului de conexiuni deschise** simultan la sursele de date.

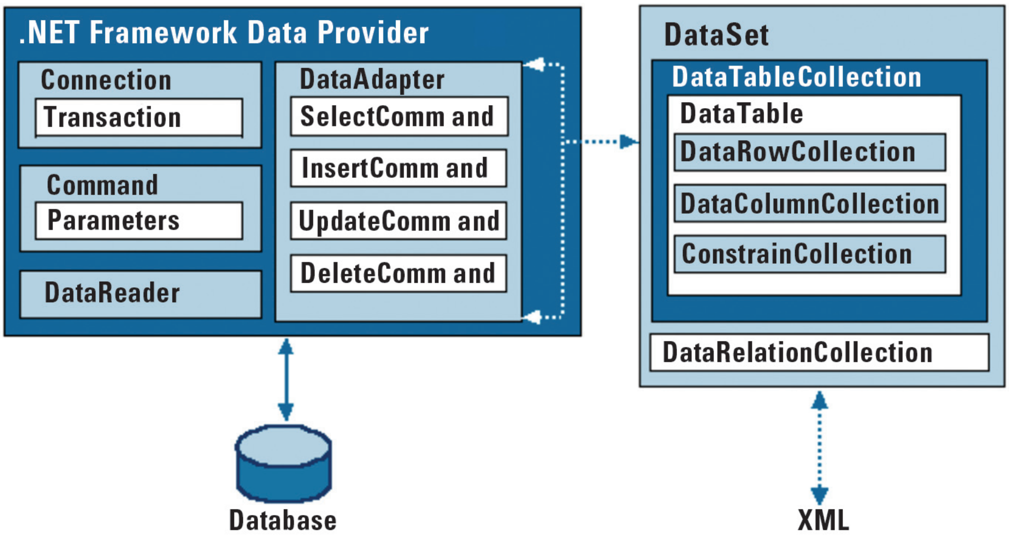
ADO.NET oferă instrumentele de utilizare și reprezentare **XML** pentru transferul datelor între aplicații și surse de date, furnizând o reprezentare comună a datelor, ceea ce permite accesarea datelor din diferite surse de diferite tipuri și prelucrarea lor ca entități, fără să fie necesar să convertim explicit datele în format XML sau invers.

Aceste caracteristici sunt determinate în stabilirea beneficiilor furnizate de ADO.NET:

- **Interoperabilitate.** ADO.NET poate interacționa ușor cu orice componentă care suportă XML.
- **Durabilitate.** ADO.NET permite dezvoltarea arhitecturii unei aplicații datorită modului de transfer a datelor între nivelele arhitecturale.
- **Programabilitate.**ADO.NET simplifică programarea pentru diferite task-uri cum ar fi comenzile SQL, ceea ce duce la o creștere a productivității și la o scădere a numărului de erori.
- **Performanță.** Nu mai este necesară conversia explicită a datelor la transferul între aplicații, fapt care duce la crește performanțelor acestora.
- **Accesibilitate** Utilizarea arhitecturii deconectate permite accesul simultan la același set de date. Reducerea numărului de conexiuni deschise simultan determină utilizarea optimă a resurselor.

6.1. Arhitectura ADO.NET

Componentele principale ale ADO.NET sunt DataSet și Data Provider. Ele au fost proiectate pentru accesarea și manipularea datelor.



6.2. Furnizori de date (Data Providers)

Din cauza existenței mai multor tipuri de surse de date este necesar ca pentru fiecare tip de protocol de comunicare să se folosească o bibliotecă specializată de clase.

.NET Framework include **SQL Server.NET Data Provider** pentru interacțiune cu Microsoft SQL Server, **Oracle Data Provider** pentru bazele de date Oracle și **OLE DB Data Provider** pentru accesarea bazelor de date ce utilitează tehnologia OLE DB pentru expunerea datelor (de exemplu Access, Excel sau SQL Server versiune mai veche decât 7.0)

Furnizorul de date permite unei aplicații să se conecteze la sursa de date, execută comenzi și salvează rezultate. Fiecare furnizor de date cuprinde componentele **Connection**, **Command**, **DataReader** și **DataAdapter**.

6.3.Connection.

Înainte de orice operație cu o sursă de date externă, trebuie realizată o conexiune (legătură) cu acea sursă. Clasele din categoria Connection (*SQLConnection*, *OleDbConnection* etc.) conțin date referitoare la sursa de date (locația, numele și parola contului de acces, etc.), metode pentru deschiderea/închiderea conexiunii, pornirea unei tranzacții etc. Aceste clase se găsesc în subspații (*SqlClient*, *OleDb* etc.) ale spațiului *System.Data*. În plus, ele implementează interfața *IdbConnection.*

Pentru deschiderea unei conexiuni prin program se poate instanția un obiect de tip conexiune, precizându-i ca parametru un șir de caractere conținând date despre conexiune.

6.3.1. Exemple de conectare

Ex.1) conectare la o sursă de date SQL

```
using System.Data.SqlClient;
SqlConnection co = new SqlConnection();
co.ConnectionString = "Data Source=localhost; User ID=profesor;pwd=info; Initial
Catalog=Orar";
co.Open();
```

Ex.2) conectare la o sursă de date SQL

```
using System.Data.SqlClient;
SqlConnection co =
new SqlConnection(@"Data Source=serverBD;Database=scoala;User
ID=elev;Password=madonna");
co.Open();
```

Ex.3) conectare la o sursă de date Access

```
using System.Data.OleDb;
OleDbConnection co =
new OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Date\scoala.mdb");
co.Open();
```

6.3.2. Proprietăţi

a) **ConnectionString** (String, cu accesori de tip **get** şi **set**) defineşte un şir care permite identificarea tipului şi sursei de date la care se face conectarea şi eventual contul şi parola de acces. Conţine lista de parametri necesarii conectării sub forma *parametru=valoare*, separaţi prin ;.

Parametru	**Descriere**
Provider	Specifică furnizorul de date pentru conectarea la sursa de date. Acest furnizor trebuie precizat doar dacă se foloseşte OLE DB .NET Data Provider, şi nu se specifică pentru conectare la SQL Server.
Data Source	Identifică serverul, care poate fi local, un domeniu sau o adresa IP.
Initial Catalog	specifică numele bazei de date. Baza de date trebuie să se găsească pe serverul dat în Data Source
Integrated Security[32]	Logarea se face cu user-ul configurat pentru Windows.
User ID	Numele unui user care are acces de logare pe server
Password	Parola corespunzătoare ID-ului specificat.

b) **ConnectionTimeout** (int, cu accesor de tip **get**): specifică numărul de secunde pentru care un obiect de conexiune poate să aştepte pentru realizarea conectării la

[32] User Id şi Password pot înlocui parametrul Integrated Security

server înainte de a se genera o excepție. (implicit 15). Se poate specifica o valoare diferită de 15 în ConnectionString folosind parametrul Connect Timeout, Valoarea Timeout=0 specifică așteptare nelimitată.

Ex.)
```
using System.Data.SqlClient;
        SqlConnection cn = new SqlConnection("Data Source=serverBD;
        Database=scoala;User ID=elev;Password=madonna; Connect Timeout=30");
```

c) **`Database`** (string, read-only): returnează numele bazei de date la care s—a făcut conectarea. Este necesară pentru a arăta unui utilizator care este baza de date pe care se face operarea

d) **`Provider`** (de tip string, read-only): returnează furnizorul de date

e) **`ServerVersion`** (string, read-only): returnează versiunea de server la care s-a făcut conectarea.

f) **`State`** (enumerare de componente *ConnectionState,* read-only): returnează starea curentă a conexiunii. Valorile posibile: *Broken*, *Closed*, *Connecting*, *Executing*, *Fetching*, *Open*.

6.3.3. Metode

a) **`Open()`**: deschide o conexiune la baza de date

b) **`Close()`** și **`Dispose()`**: închid conexiunea și eliberează toate resursele alocate pentru ea

c) **`BeginTransaction()`**: pentru executarea unei tranzacții pe baza de date; la sfârșit se apelează **`Commit()`** sau **`Rollback()`**.

d) **`ChangeDatabase()`**: se modifică baza de date la care se vor face conexiunile. Noua bază de date trebuie să existe pe același server ca și precedenta.

e) **`CreateCommand()`**: creează o comandă (un obiect de tip *Command*) validă asociată conexiunii curente.

6.3.4. Evenimente

a) **`StateChange`**: apare atunci când se schimbă starea conexiunii. Handlerul corespunzător (de tipul delegat *StateChangeEventHandler*) spune între ce stări s-a făcut tranziția.

b) **`InfoMessage`**: apare când furnizorul trimite un avertisment sau un mesaj către client.

6.4. Command

Clasele din categoria Command (*SQLCommand*, *OleDbCommand* etc.) conțin date referitoare la o comandă SQL (SELECT, INSERT, DELETE, UPDATE) și metode pentru executarea unei comenzi sau a unor proceduri stocate. Aceste clase implementează interfața *IDbCommand*. Ca urmare a interogării unei baze de date se obțin obiecte din categoriile *DataReader* sau *DataSet*. O comandă se poate executa numai după ce s-a stabilit o conxiune cu baza de date corespunzătoare.

6.4.1. Proprietăți

a) **CommandText** (String): conține comanda SQL sau numele procedurii stocate care se execută pe sursa de date.

b) **CommandTimeout** (int): reprezintă numărul de secunde care trebuie să fie așteptat pentru executarea comenzii. Dacă se depășeste acest timp, atunci se generează o excepție.

c) **CommandType** (enumerare de componente de tip *CommandType*): reprezintă tipul de comandă care se execută pe sursa de date. Valorile pot fi: StoredProcedure (apel de procedură stocată), Text (comandă SQL obișnuită), TableDirect (numai pentru OleDb)

d) **Connection** (System. Data. [Provider].PrefixConnection): conține obiectul de tip conexiune folosit pentru legarea la sursa de date.

e) **Parameters** (System.Data.[Provider].PrefixParameterCollection): returnează o colecție de parametri care s-au transmis comenzii.

f) **Transaction** (System.Data.[Provider].PrefixTransaction): permite accesul la obiectul de tip tranzacție care se cere a fi executat pe sursa de date.

6.4.2. Metode

a) Constructori:

SqlCommand()

SqlCommand(string CommandText)

SqlCommand(string CommandText, SqlConnection con)

SqlCommand(string CommandText,SqlConnection con,SqlTransaction trans)

b) **Cancel()** oprește o comandă aflată în executare.

c) **Dispose()** distruge obiectul comandă.

d) **ExecuteNonQuery()** execută o comandă care nu returnează un set de date din baza de date; dacă comanda a fost de tip INSERT, UPDATE, DELETE, se returnează numărul de înregistrări afectate.

Exemplu:

```
SqlCommand cmd = new SqlCommand();
  cmd.CommandText = "DELETE FROM elevi WHERE nume = 'BARBU'";
  cmd.Connection = con;
  Console.WriteLine(cmd.ExecuteNonQuery().ToString());
                                        //câte înreg. s-au sters
```

e) **ExecuteReader()** execută comanda și returnează un obiect de tip DataReader.

Exemplu

Se obține conținutul tabelei elevi într-un obiect de tip SqlDataReader.

```
SqlCommand cmd = new SqlCommand("SELECT * FROM elevi",con);
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read()) { Console.WriteLine("{0} - {1}",
```

```
reader.GetString(0),reader.GetString(1));
        }
        reader.Close();
```

Metoda ExecuteReader() mai are un argument opțional de tip enumerare, **CommandBehavior**, care descrie rezultatele și efectul asupra bazei de date:

CloseConnection (conexiunea este închisă atunci când obiectul DataReader este închis),

KeyInfo (returneză informație despre coloane și cheia primară),

SchemaOnly (returneză doar informație despre coloane),

SequentialAccess (pentru manevrarea valorilor binare cu GetChars() sau GetBytes()),

SingleResult (se returnează un singur set de rezultate),

SingleRow (se returnează o singură linie).

f) **ExecuteScalar()** execută comanda și returnează valoarea primei coloane de pe primul rând a setului de date rezultat; folosit pentru obținerea unor rezultate statistice.

Exemplu:

```
SqlCommand cmd = new SqlCommand("SELECT COUNT(*) FROM elevi",con);
                 SqlDataReader reader = cmd.ExecuteScalar();
                 Console.WriteLine(reader.GetString(0));
```

f) **ExecuteXmlReader()** returnează un obiect de tipul XmlReader obținut prin interogare

Exemplu:

```
SqlCommand CMD=
   new SqlCommand("SELECT * FROM elevi FOR XML MATE,EXAMEN", con);
   System.Xml.XmlReader myXR = CMD.ExecuteXmlReader();
```

Obiectele de tip SQLCommand pot fi utilizate într-un scenariu ce presupune deconectarea de la sursa de date dar și în operații elementare care presupun obținerea unor rezultate imediate.

Vom exemplifica utilizarea obiectelor de tip Command în operații ce corespund acestui caz.

Presupunem că am stabilit conexiunea:

```
using System.Data.SqlClient;
SqlConnection conn =
new SqlConnection(@"Data Source=serverBD;Database=MAGAZIN;User ID=adm;Password=eu");
conn.Open();
```

și că tabela PRODUSE are câmpurile ID_PRODUS, DENUMIRE_PRODUS, DESCRIERE
Instanțierea unui obiect de tip SQLCommnand

```
SqlCommand cmd = new SqlCommand("select DENUMIRE_PRODUS from PRODUSE", conn);
```

conține un string ce precizează comanda care se execută și o referință către obiectul SQLConnection.

6.4.3. Interogarea datelor.

Pentru extragerea datelor cu ajutorul unui obiect SqlCommand trebuie să utilizăm metoda ExecuteReader care returnează un obiect SqlDataReader.

```
  //  Instantiem o comandă cu o cerere si precizam conexiunea
  SqlCommand cmd = new SqlCommand("select DENUMIRE_PRODUS from PRODUSE", conn);
// Obtinem rezultatul cererii
SqlDataReader rdr = cmd.ExecuteReader();
```

6.4.4. Inserarea datelor.

Pentru a insera date într-o bază de date utilizăm metoda ExecuteNonQuery a obiectului SqlCommand.

```
  // Sirul care păstrează comanda de inserare
 string insertString = @"insert into PRODUSE(DENUMIRE_PRODUS, DESCRIERE)
                                                       values ('Barbie', 'papusa')";
 // Instantiem o comandă cu această cerere si precizăm conexiunea
 SqlCommand cmd = new SqlCommand(insertString, conn);
 // Apelăm metoda ExecuteNonQuery pentru a executa comanda
 cmd.ExecuteNonQuery();
```

Facem observația că am specficat explicit numai coloanele DENUMIRE_PRODUS și DESCRIERE. Tabela PRODUSE are cheia primară ID_PRODUS. Valoarea acestui câmp va fi atribuită de SQL Server. Dacă încercăm să adăugăm o valoare atunci va fi generată o excepție.

6.4.5. Actualizarea datelor.

```
  // Sirul care păstrează comanda de actualizare
 string updateString = @"update PRODUSE set DENUMIRE_PRODUS = 'Locomotiva Thomas'
    where DENUMIRE_PRODUS = 'Thomas'";
 //  Instantiem o nouă comandă fără să precizăm conexiunea
 SqlCommand cmd = new SqlCommand(updateString);
 //  Stabilim conexiunea
```

```
cmd.Connection = conn;33
// Apelăm ExecuteNonQuery pentru executarea comenzii
cmd.ExecuteNonQuery();
```

6.4.6. Ștergerea datelor.

Se utilizează aceeași metodă ExecuteNonQuery.

```
  // sirul care păstrează comanda de ştergere
string deleteString = @"delete from PRODUSE where DENUMIRE_PRODUS = 'Barbie'";
// Instanţiem o comandă
SqlCommand cmd = new SqlCommand();34
// Setăm proprietatea CommandText
cmd.CommandText = deleteString;
// Setăm proprietatea Connection
cmd.Connection = conn;
// . Executăm comanda
cmd.ExecuteNonQuery();
```

Câteodată avem nevoie să obținem din baza de date o singură valoare, care poate fi o sumă, o medie sau alt rezultat al unei funcții agregat. O alegere ineficientă ar fi utilizarea metodei ExecuteReader și apoi calculul valorii. În acest caz, cea mai bună alegere este să lucrăm direct asupra bazei de date și să obținem această valoare.

```
  // Instantiem o comandă nouă
SqlCommand cmd = new SqlCommand("select count(*) from PRODUSE", conn);
// Executăm comanda si obtinem valoarea
int count = (int)cmd.ExecuteScalar();35
```

Exerciții:

1) Realizați o conexiune la baza de date MAGAZIN și afișați ID-urile produselor.

```
  using System;
  using System.Data;
  using System.Data.SqlClient;
class ADO1
{
    static void Main()
    {  SqlConnection conn = new SqlConnection(
          "Data Source=(local);Initial Catalog=MAGAZIN;Integrated Security=SSPI");
         SqlDataReader rdr = null;
         try
         {  conn.Open();
              SqlCommand cmd = new SqlCommand("select * from PRODUSE", conn);
```

[33] Am fi putut folosi același constructor ca la Insert. Acest exemplu demonstrează că putem schimba oricând obiectul connection asignat unei comenzi

[34] În acest exemplu am ales să aplelă constructorul SqlCommand fără parametri, pentru a exemplifica cum putem stabili explicit conexiunea și comanda

[35] Este necesară conversia întrucât rezultatul returnat de ExecuteScalar este de tip object

```
                rdr = cmd.ExecuteReader();
                while (rdr.Read()) { Console.WriteLine(rdr[0]);}
            }
            finally
            {   if (rdr != null) {rdr.Close();}
                if (conn != null){conn.Close();}
            }
        }
}
```

2) Realizați funcții care să implementeze operațiile elementare asupra unei baze de date și verificați funcționalitatea lor.

```
    using System;
    using System.Data;
    using System.Data.SqlClient;
   class ADO2
 {
     SqlConnection conn;

     public ADO2()
     { conn = new SqlConnection("Data Source=(local);Initial
Catalog=MAGAZIN;Integrated Security=SSPI");
     }
     static void Main()
     {
         ADO2 scd = new ADO2();
            Console.WriteLine("Produse aflate în magazin înainte de Insert");
         scd.ReadData();scd.Insertdata();
         Console.WriteLine("Produse aflate în magazin dupa Insert");
            scd.ReadData();scd.UpdateData();
            Console.WriteLine("Produse aflate în magazin dupa Update");
            scd.ReadData();scd.DeleteData();
            Console.WriteLine("Categories After Delete");
            scd.ReadData();
         int number_inregistrari = scd.GetNumberOfRecords();
         Console.WriteLine("Numarul de inregistrari: {0}", numar_inregistrari);
     }
    public void ReadData()
     { SqlDataReader rdr = null;
          try
          {conn.Open();
            SqlCommand cmd = new SqlCommand("select DENUMIRE_PRODUS from PRODUSE",
conn);
            rdr = cmd.ExecuteReader();
             while (rdr.Read()) {Console.WriteLine(rdr[0]);}
          }
          finally
          { if (rdr != null){rdr.Close();}
```

```
            if (conn != null){conn.Close();}
         }
     }
   public void Insertdata()
     {try
         {conn.Open();
           string insertString = @"insert into PRODUSE(DENUMIRE_PRODUS, DESCRIERE)
                 values ('SCOOBY', 'jucarie de plus')";
            SqlCommand cmd = new SqlCommand(insertString, conn);
            cmd.ExecuteNonQuery();
         }
         finally
         {if (conn != null){conn.Close();}
         }
     }
       public void UpdateData()
     {
         try
         {conn.Open();
           string updateString = @"update PRODUSE set DENUMIRE_PRODUS = 'SCOOBY DOO'
                 where DENUMIRE_PRODUS = 'SCOOBY'";
             SqlCommand cmd = new SqlCommand(updateString);
             cmd.Connection = conn;
             cmd.ExecuteNonQuery();
        }
         finally
         {if (conn != null){conn.Close();}
         }
     }
      public void DeleteData()
     { try
         { conn.Open();
           string deleteString = @"delete from PRODUSE where DENUMIRE_PRODUS = 'BAR-
BIE'";
           SqlCommand cmd = new SqlCommand();
            cmd.CommandText = deleteString;
            cmd.Connection = conn;
            cmd.ExecuteNonQuery();
         }
         finally
         {if (conn != null){conn.Close();}}
     }
       public int GetNumberOfRecords()
     {
         int count = -1;
         try
         { conn.Open();
            SqlCommand cmd = new SqlCommand("select count(*) from Produse", conn);
            count = (int)cmd.ExecuteScalar();
         }
```

```
          finally
          {if (conn != null){conn.Close();}
          }
          return count;
     }
}
```

6.5. DataReader

Datele pot fi explorate în mod conectat (cu ajutorul unor obiecte din categoria *DataReader*), sau pot fi preluate de la sursă (dintr-un obiect din categoria *DataAdapter*) și înglobate în aplicația curentă (sub forma unui obiect din categoria *DataSet*).

Clasele **DataReader** permit parcurgerea într-un singur sens a sursei de date, fără posibilitate de modificare a datelor la sursă. Dacă se dorește modificarea datelor la sursă, se va utiliza ansamblul *DataAdapter* + *DataSet*.

Datorita faptului că citește doar înainte (*forward-only*) permite acestui tip de date să fie foarte rapid în citire. Overhead-ul asociat este foarte mic (overhead generat cu inspectarea rezultatului și a scrierii în baza de date). Dacă într-o aplicație este nevoie doar de informații care vor fi citite o singura dată, sau rezultatul unei interogări este prea mare ca sa fie reținut în memorie (caching) DataReader este soluția cea mai bună.

Un obiect *DataReader* nu are constructor[36], ci se obține cu ajutorul unui obiect de tip *Command* și prin apelul metodei ExecuteReader() (vezi exercițiile de la capitolul anterior). Evident, pe toată durata lucrului cu un obiect de tip *DataReader*, conexiunea trebuie să fie activă. Toate clasele *DataReader* (*SqlDataReader*, *OleDbDataReader* etc.) implementează interfața *IDataReader*.

6.5.1. Proprietăți:

a) **`IsClosed`** `(boolean, read-only)`- returneză true dacă obiectul este deschis si fals altfel
b) **`HasRows`** (boolean,read-only) - verifică dacă reader-ul conține cel puțin o înregistrare
c) **`Item`** (indexator de câmpuri)
d) **`FieldCount`** - returnează numărul de câmpuri din înregistrarea curentă

6.5.2. Metode:

a) **`Close()`** închidere obiectului și eliberarea resurselor; trebuie să preceadă închiderea conexiunii.
b) **`GetBoolean()`**, **`GetByte()`**, **`GetChar()`**, **`GetDateTime()`**, **`GetDecimal()`**, **`GetDouble()`**, **`GetFloat()`**, **`GetInt16()`**, **`GetInt32()`**, **`GetInt64()`**, **`GetValue()`**, **`GetString()`** returnează valoarea unui câmp specificat, din înergistrarea curentă

[36] Dacă pentru instantiere este folosit operatorul new veți obține un obiect cu care nu puteți face nimic pentru că nu are o conexiune și o comandă atașate.

c) **`GetBytes()`**, **`GetChars()`** citirea unor octeți/caractere dintr—un câmp de date binar
d) **`GetDataTypeName()`**, **`GetName()`** returnează tipul/numele câmpului specificat
e) **`IsDBNull()`** returnează true dacă în câmpul specificat prin index este o valoare NULL
f) **`NextResult()`** determină trecerea la următorul rezultat stocat în obiect (vezi exemplul)
g) **`Read()`** determină trecerea la următoarea înregistrare, returnând *false* numai dacă aceasta nu există; de reținut că inițial poziția curentă este **înaintea** primei înregistrări.

DataReader obține datele într-un stream secvențial. Pentru a citi aceste informații trebuie apelată metoda Read; aceasta citeste un singur rând din tabelul rezultat. Metoda clasică de a citi informația dintr-un DataReader este de a itera într-o buclă while.

Ex.1)

```
SqlCommand cmd=new SqlCommand("select * from elevi;select * from profi", conn );
conn.Open ();
SqlDataReader reader = cmd.ExecuteReader ();
do {  while ( reader.Read () )
                {Console.WriteLine ( "{0}\t\t{1}", reader[0], reader[1] );}
      } while ( reader.NextResult () );
```

DataReader implementează și indexatori (în exemplul anterior am afișat primele coloane folosind indexatori numerici).

Nu este foarte clar pentru cineva care citește codul care sunt coloanele afișate decât dacă s-a uitat și în baza de date. Din aceasta cauză este preferată utilizarea indexatorilor de tipul string. Valoarea indexului trebuie să fie numele coloanei din tabelul rezultat.

Indiferent că se folosește un index numeric sau unul de tipul string indexatorii întorc totdeauna un obiect de tipul object fiind necesară conversia.

Exemplu: Codul

```
SqlCommand cmd = new SqlCommand("select * from PRODUSE", conn);
   rdr = cmd.ExecuteReader();
   while (rdr.Read()) {Console.WriteLine(rdr[0]);}
```

este echivalent cu

```
SqlCommand cmd = new SqlCommand("select * from PRODUSE", conn);
   rdr = cmd.ExecuteReader();
      while (rdr.Read()){Console.WriteLine (rdr["ID_PRODUS"]);
```

Exercițiu. Afișați conținutul tabelei PRODUSE utilizând DataReader.

```
using System;
using System.Data;
using System.Data.SqlClient;

class ADO3
{ static void Main()
            {ADO3 rd = new ADO3();
             rd.SimpleRead();
            }

public void SimpleRead()
{ SqlDataReader rdr = null;
  SqlConnection conn = new SqlConnection(
  "Data Source=(local);Initial Catalog=MAGAZIN;Integrated Security=SSPI");
  SqlCommand cmd  = new SqlCommand("select * from PRODUSE", conn);
  try { conn.Open();
          rdr = cmd.ExecuteReader();
          Console.WriteLine("DENUMIRE PRODUS DESCRIERE");
          while (rdr.Read()){string den = (string)rdr["DENUMIRE_PRODUS"];
                   string descr = (string)rdr["DESCRIERE"];
                                  Console.Write("{0,-20}", den);
                   Console.Write("{0,-30}", descr);
                          Console.WriteLine();
                         }
        }
 finally {if (rdr != null){rdr.Close();}
             if (conn != null){conn.Close();}
           }
    }
}
```

6.6. DataAdapter

Folosirea combinată a obiectelor **DataAdapter** și **DataSet** permite operații de selectare, ștergere, modificare și adăugare la baza de date. Clasele *DataAdapter* generează obiecte care funcționează ca o interfață între sursa de date și obiectele *DataSet* interne aplicației, permițând prelucrări pe baza de date. Ele gestionează automat conexiunea cu baza de date astfel încât conexiunea să se facă numai atunci când este imperios necesar.

Un obiect *DataSet* este de fapt un set de tabele relaționate. Folosește serviciile unui obiect *DataAdapter* pentru a-și procura datele și trimite modificările înapoi către baza de date. Datele sunt stocate de un *DataSet* în format XML, același folosit și pentru transferul datelor.

În exemplul următor se preiau datele din tablele elevi și profi:

```
SqlDataAdapter de=new SqlDataAdapter("SELECT nume,clasa FROM elevi", conn);
de.Fill(ds,"Elevi");//transferă datele în datasetul ds sub forma unei tabele locale
numite elevi
SqlDataAdapter dp=new SqlDataAdapter("SELECT nume, clasdir FROM profi",conn);
```

```
dp.Fill(ds,"Profi");//transferă datele în datasetul ds sub forma unei tabele locale
numite profi
```

6.6.1. Proprietăți

a) **DeleteCommand**, **InsertCommand**, **SelectCommand**, **UpdateCommand** (Command), conțin comenzile ce se execută pentru selectarea sau modificarea datelor în sursa de date.

b) **MissingSchemaAction** (enumerare) determină ce se face atunci când datele aduse nu se potrivesc peste schema tablei în care sunt depuse. Poate avea următoarele valori:
Add - implicit, DataAdapter adaugă coloana la schema tablei
AddWithKey — se adugă coloana și informații relativ la cheia primară
Ignore - se ignoră lipsa coloanei respective, ceea ce duce la pierdere de date
Error - se generează o excepție de tipul *InvalidOperationException*.

6.6.2. Metode

Constructori:**SqlDataAdapter()|SqlDataAdapter(obiect_comanda)|**
SqlDataAdapter(string_comanda, conexiune);

a) **Fill()** permite umplerea unei tabele dintr—un obiect *DataSet* cu date. Permite specificarea obiectului *DataSet* în care se depun datele, eventual a numelui tablei din acest *DataSet*, numărul de înregistrare cu care să se înceapă popularea (prima având indicele 0) și numărul de înregistrări care urmează a fi aduse.

a) **Update()** permite transmiterea modificărilor efectuate într—un *DataSet* către baza de date.

6.7. DataSet

Un *DataSet* este format din *Tables* (colecție formată din obiecte de tip *DataTable*; *DataTable* este compus la rândul lui dintr-o colecție de *DataRow* și *DataColumn*), *Relations* (colecție de obiecte de tip *DataRelation* pentru memorarea legăturilor părinte—copil) și *ExtendedProperties* ce conține proprietăți definite de utilizator.

Scenariul uzual de lucru cu datele dintr-o tabelă conține următoarele etape:

- popularea succesivă a unui *DataSet* prin intermediul unuia sau mai multor obiecte *DataAdapter*, apelând metoda *Fill* (vezi exemplul de mai sus)
- procesarea datelor din *DataSet* folosind numele tabelelor stabilite la umplere, ds.Tables["elevi"], sau indexarea acestora, ds.Tables[0], ds.Tables[1]
- actualizarea datelor prin obiecte comandă corespunzătoare operațiilor INSERT, UPDATE și DELETE. Un obiect *CommandBuilder* poate construi automat o combinație de comenzi ce reflectă modificările efectuate.

Așadar, DataAdapter deschide o conexiune doar atunci când este nevoie și o inchide imediat aceasta nu mai este necesară.

De exemplu DataAdapter realizează următoarele operațiuni atunci când trebuie sa populeze un DataSet:*deschide conexiunea, populeaza DataSet-ul,închide conexiunea* și urmatoărele operațiuni atunci când trebuie sa facă update pe baza de date: *deschide conexiunea, scrie modificarile din DataSet in baza de date,inchide conexiunea*. Intre operațiunea de populare a DataSet-ului și cea de update conexiunile sunt inchise. Intre aceste operații în DataSet se poate scrie sau citi.

Crearea unui obiect de tipul DataSet se face folosind operatorul new.

Exemplu. `DataSet dsProduse = new DataSet ();`

Constructorul unui DataSet nu necesită parametri. Există totuși o supraîncărcare a acestuia care primește ca parametru un string și este folosit atunci cand trebuie să se facă o serializare a datelor într-un fisier XML. In exemplul anterior avem un DataSet gol și avem nevoie de un DataAdapter pentru a-l popula.

Un obiect DataAdapter conține mai multe obiecte Command (pentru inserare, update, delete și select) și un obiect Connection pentru a citi și scrie date.

În exemplul următor construim un obiect de tipul DataAdapter, daProd. Comanda SQL specifică cu ce date va fi populat un DataSet, iar conexiunea conn trebuie să fi fost creată anterior, dar nu și deschisă. DataAdapter-ul va deschide conexiunea la apelul metodelor *Fill* și *Update*.

```
SqlDataAdapter daProd =
new SqlDataAdapter ("SELECT ID_PRODUS, DENUMIRE_PRODUS FROM PRODUSE", conn);
```

Prin intermediul constructorului putem instanția doar comanda de interogare. Instanțierea celorlalte se face fie prin intermediul proprietătilor pe care le expune DataAdapter, fie folosind obiecte de tipul CommandBuilder.

SqlCommandBuilder cmdBldr = new SqlCommandBuilder (daProd);

La inițializarea unui CommandBuilder se aplelează un constructor care primește ca parametru un adapter, pentru care vor fi construite comenzile. SqlCommandBuilder are nu poate construi decât comenzi simple și care se aplică unui singur tabel. Atunci cand trebui ca sa facem comenzi care vor folosi mai multe tabele este recomandată construirea separată a comnezilor și apoi atasarea lor adapterului folosind proprietăți.

Popularea DataSet-ului se face după ce am construit cele două instanțe:

```
daProd.Fill (dsProduse, "PRODUSE");
```

În exemplul următor va fi populat DataSet-ul dsProduse. Cel de-al doilea parametru (string) reprezintă numele tabelului (nu numele tabelului din baza de date, ci al tabelului rezultat în DataSet) care va fi creat. Scopul acestui nume este identificarea ulterioară a tabelului. În cazul în care nu sunt specificate numele tabelelor, acestea vor fi adăugate în DataSet sub numele *Table1*, *Table2*, ...

Un DataSet poate fi folosit ca sursă de date pentru un DataGrid din Windows Forms sau ASP.Net .

Exemplu.

```
DataGrid dgProduse = new DataGrid();
       dgProduse.DataSource = dsProduse;
      dgProduse.DataMembers = "PRODUSE";37
```

După ce au fost făcute modificări într-un DataSet acestea trebuie scrise și în baza de date. Actualizarea se face prin apelul metodei Update.

daProd.Update (dsProduse, "PRODUSE");

6.8. SqlParameter

Atunci când lucrăm cu bazele de date avem nevoie, de cele mai multe ori să filtrați rezultatul după diverse criterii. De obicei acest lucru se face în funcție de niște criterii pe care utilizatorul le specifică (ex: vreți să vedeți doar păpușile Barbie).

Cea mai simplă metodă de filtrare a rezultatelor este să construim dinamic string-ul SqlCommand dar această metoda nu este recomandată deoarece poate afecta baza de date (ex. Accesarea informațiilor confidențiale).

Dacă folosim interogări cu parametri atunci orice valoare pusă într-un parametru nu va fi tratată drept cod SQL, ci ca valoare a unui câmp, făcând aplicația mai sigură.

Pentru a folosi interogări cu parametri trebuie să:

a) construim string-ul pentru SqlCommand folosind parametri;

```
Ex. SqlCommand cmd =
  new SqlCommand("SELECT * FROM PRODUSE WHERE DENUMIRE = @den", conn);38
```

b) construim un obiect SqlParameter asignând valorile corespunzătoare;

```
Ex. SqlParameter param = new SqlParameter();
      param.ParameterName = "@Cden";
      param.Value = sir;
```

c) adăugați obiectul SqlParameter la obiectul SqlCommand, folosind proprietatea Parameters.

Ex. `cmd.Parameters.Add(param);`

[37] Se pot afișa mai multe tabele dintr-un DataSet, semnul "+" permițându-i utilizatorului să aleaga care tabel să fie afișat. Pentru a suprima afișarea acelui semn "+" setăm proprietatea DataMembers pe numele tabelului care va fi afișat. Numele tabelului este același care l-am folosit ca parametru în apelul metodei Fill.

[38] Atunci când comanda va fi executată @den va fi înlocuit cu valoarea aflată în obiectul SqlParameter atașat. Dacă nu asociem o instanță de tipul SqlParameter pentru un parametru din string-ul de interogare sau avem mai multe instanțe SqlParameter pentru un parametru vom obține o eroare la rulare

6.9. Proceduri Stocate (Stored Procedures)

O procedură stocată este o secvență de instrucțiuni SQL, salvată în baza de date, care poate fi apelată de aplicații diferite. Sql Server compilează procedurile stocate, ceea ce crește eficiența utilizării lor. De asemenea, procedurile stocate pot avea parametri.

O procedură stocată poate fi apelată folosind obiectul SqlCommand:

```
SqlCommand cmd = new SqlCommand("StoredProcedure1", conn);
cmd.CommandType = CommandType.StoredProcedure;  //Tipul obiectului comanda este pro-
cedura stocata
```

Primul parametru al constructorului este un șir de caractere ce reprezintă numele procedurii stocate. A doua instrucțiune de mai sus spune obiectului SqlCommand ce tip de comandă va fi executată, prin intermediul proprietății CommandType.

Exemplu:

```
            SqlCommand cmd = new SqlCommand("StoredProcedure1", conn);
            cmd.CommandType = CommandType.StoredProcedure;  //Tipul obiectului coman-
da este procedura stocata

             personDs = new DataSet();
             personDa = new SqlDataAdapter("", conn);
             personDa.SelectCommand = cmd;
        personDa.Fill(personDs, "PersonTable");
```

Apelul procedurilor stocate, parametrizate, este asemănator cu cel al interogărilor cu parametri.

```
            //Obiect Comanda, in care primul parametru este numele procedurii stocate
             SqlCommand cmd = new SqlCommand("City", conn);
            cmd.CommandType = CommandType.StoredProcedure;  //Tipul obiectului coman-
da este procedura stocata
             cmd.Parameters.Add(new SqlParameter("@City", inputCity));

             personDs = new DataSet();
             personDa = new SqlDataAdapter("", conn);
             personDa.SelectCommand = cmd;
        personDa.Fill(personDs, "PersonTable");
```

Primul argument al constructorului obiectului SqlCommand este numele procedurii stocate. Această procedură are un parametru numit @City. De aceea trebuie folosit un obiect de tip SqlParameter pentru a adauga acest parametru la obiectul de tip Command.

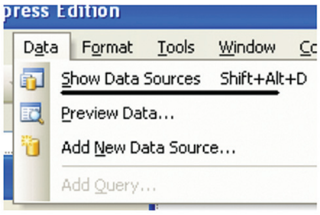
Exercițiu de sinteză. Construiți o aplicație pentru a simula gestiunea unei biblioteci școlare.

Precizări. Toate informațiile se vor afla într-o bază de date. Creati propriile structuri de date adecvate rezolvarii problemei. Utilizati Microsoft Access pentru crearea bazei de date. Inițial aplicația va afișa o formă Windows care permite selectarea operației efectuate (adăugare carte/cărți, adăugare abonat, actualizare stare carte/cărți/abonat, împrumută carte/cărți, etc.)

6.11. Proiectarea vizuală a seturilor de date

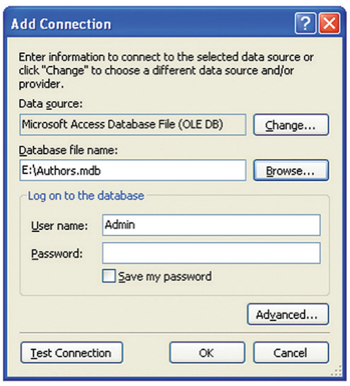
Mediul de dezvoltare Visual Studio dispune de instrumente puternice și sugestive pentru utilizarea bazelor de date în aplicații. Conceptual, în spatele unei ferestre în care lucrăm cu date preluate dintr-una sau mai multe tabele ale unei baze de date se află obiectele din categoriile *Connection*, *Command*, *DataAdapter* și *DataSet* prezentate. ”La vedere” se află controale de tip *DataGridView*, sau *TableGridView*, *BindingNavigator* etc.

Meniul *Data* și fereastra auxiliară *Data Sources* ne sunt foarte utile în lucrul cu surse de date externe.

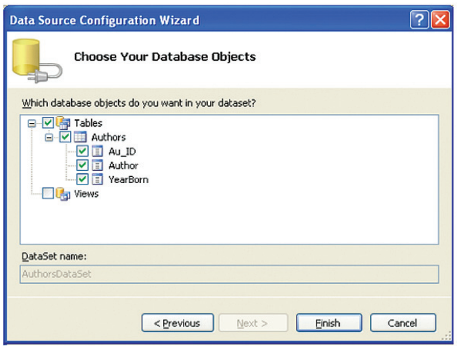


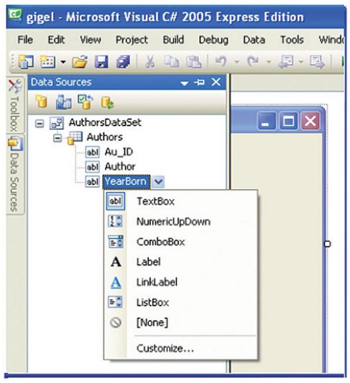
Să urmărim un scenariu de realizare a unei aplicații simple cu o fereastră în care putem vizualiza date dintr-o tabelă, putem naviga, putem modifica sau șterge înregistrări.

- Inițiem adăugarea unei surse de date (*Add New Source*)
- Configurăm cu atenție (asistați de ”vrăjitor”) conexiunea cu o sursă de tip SQL sau Access; figura surprinde elemente de conectare la o bază de date Access, numită Authors, bază stocată pe hard-discul local.
- Selectăm tabelele care ne interesează din baza de date și câmpurile din cadrul tabelei ce vor fi reținute în *TableAdapter* (din categoria *DataAdapter*)

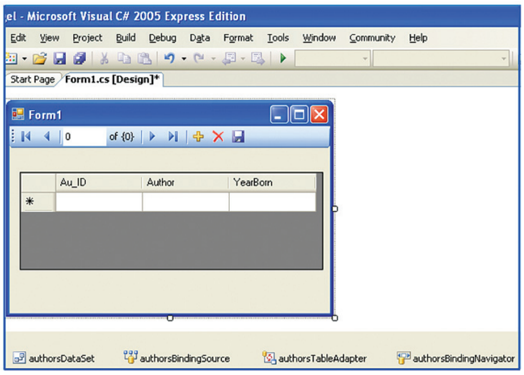


- Când operațiunea se încheie, date relative la baza de date la care ne-am conectat sunt integrate în proiect și pictograma, ca și structura bazei de date, apar în fereastra *Data Source*



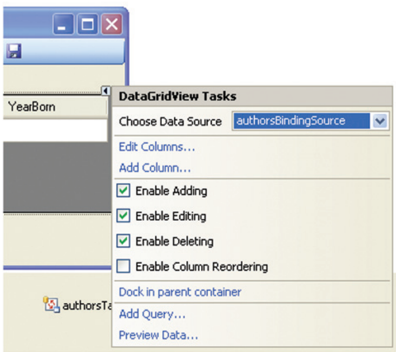


- Prin "tragerea" unor obiecte din fereastra *Data Sources* în fereastra noastră nouă, se creează automat obiecte specifice. În partea de jos a figurii se pot observa obiectele de tip *Dataset*, *TableAdapter*, *BindingSource*, *BindingNavigator* și, în fereastră, *TableGridView*



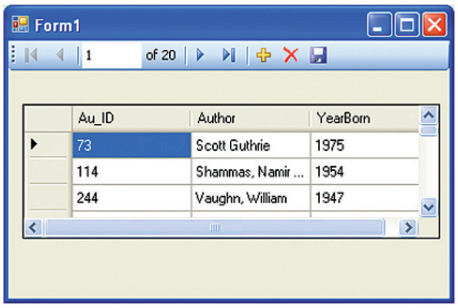
BindingNavigator este un tip ce permite, prin instanțiere, construirea barei de navigare care facilitează operații de deplasare, editare, ștergere și adăugare în tabel.

Să observăm că reprezentarea vizuală a fiecărui obiect este înzestrată cu o săgetă în partea de sus, în dreapta. Un clic pe această săgeată activează un meniu contextual cu lista principalelor operații ce se pot efectua cu obiectul respectiv.



Meniul contextual asociat grilei în care vor fi vizualizate datele permite configurarea modului de lucru cu grila (sursa de date, operațiile permise și altele).

În timpul rulării aplicației, bara de navigare și elementele vizuale ale grilei permit operațiile de bază cu înregistrările bazei de date. Operațiile care modifică baza de date trebuie să fie definitivate prin salvarea noilor date .



BIBLIOGRAFIE

- Marshall Donis, Programming Microsoft Visual C# 2005: The Language, Microsoft Press 2006, ISBN:0735621810
- Pelland Patrice, Build a Program NOW, Microsoft Visual C# 2005 Express Edition, Microsoft Press 2006,
- LearnVisualStudio.NET http://www.learnvisualstudio.net resurse educaționale gratuite sub forma de filme
- Harris Andy, Microsoft C# Programming for the Absolute Beginner, Premier Press 2002, ISBN: 1?931841?16?0
- Wright Peter, Beginning Visual C# 2005 Express Edition: From Novice to Professional, Apress 2006, ISBN-13 (pbk): 978-1-59059-549-7, ISBN-10 (pbk): 1-59059-549-1
- Liberty Jesse, Programming C#, Second Edition, O'REILLY 2002, ISBN 10: 0-596-00309-9, ISBN 13:9780596003098
- Solis Daniel, Illustrated C# 2005, Apress 2006, ISBN-13 (pbk): 978-1-59059-723-1, ISBN-10 (pbk): 1-59059-723-0
- Rasheed Faraz, C# School, Synchron Data 2005-2006, http://www.programmersheaven.com/2/CSharpBook
- Schneider Robert, Microsoft® SQL Server™ 2005 Express Edition For Dummies®, Wiley Publishing 2006, ISBN-13: 978-0-7645-9927-9, ISBN-10: 0-7645-9927-5
- Bradley L. Jones, Sams Teach Yourself the C# Language in 21 Days, SAMS, 2004, International Standard Book Number: 0-672-32546-2
- Michel de Camplain, Brian G. Patrik, C# 2.0: Practical Guide for Programmers, Elsevier, 2005, ISBN: 0-12-167451-7
- Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner, Allen Jones, Professional C# 2005, Wiley Publishing, Inc., 2006, ISBN-13: 978-0-7645-7534-1 ISBN-10: 0-7645-7534-1
- Sharp John, Visual C# 2005, DUNOD 2006
- Iulian Serban, Dragos Brezoi, Tiberiu Radu, Adam Ward, GDI+ Custom Controls with Visual C# 2005, PACKT Publishing 2006, ISBN 1-904811-60-4
- Simpson Alan, Visual Web Developer Server™ 2005 Express Edition For Dummies®, Wiley Publishing 2006, ISBN-13: 978-0-7645-8360-5, ISBN-10: 0-7645-8360-3
- Hoffman Kevin, Microsoft® Visual C# 2005 Unleashed, Sams 2006, ISBN-10: 0-672-32776-7, ISBN-13: 978-0-672-32776-6
- Popovici, Dorin Mircea și colaboratorii - Proiectare și implementare software, Editura TEORA, București, 1998
- C# Language Specification, Microsoft Corporation, 1999-2003
- C# Version 2.0 Specification May 2004, Microsoft Corporation, 1999-2005
- David Conger, Programarea în C#, editura B.I.C. ALL 2003
- Chris H. Pappas, William H. Murray C# pentru programarea Web, editura B.I.C. ALL 2004
- MCAD/MCSD -Developing XML WEB SERVICES and SERVER COMPONENTS WITH MICROSOFT
  http://www.csharp-station.com
  http://msdn2.microsoft.com
- M.E.C. Serviciul Național de Evaluare și Examinare - Ghid de evaluare, INFORMATICA și TEHNOLOGIA INFORMATIEI, Editura Aramis, București 2001
- M.E.C. Consiliul Național pentru Curriculum - Ghid metodologic pentru aplicarea programelor școlare, Editura Aramis, București 2002
- M.E.C Consiliul Național pentru curriculum - Ghid metodologic pentru Educație Tehnologică Informatică - Tehnologia Informației - Liceu teoretic - Editura Aramis, București 2001

Microsoft

Suport de curs pentru elevi

Introducere în

.Net Framework

Microsoft .net Framework

ISBN: 973-86699-5-2

Ediția 2008